AN AUTOMATED TOOL FOR FORMULATING LINEAR COVARIANCE ANALYSIS PROBLEMS

by

Andrew W. Lewin

S.B., Aeronautics and Astronautics
Massachusetts Institute of Technology, 1991

Submitted to the Department of
Aeronautics and Astronautics in Partial
Fulfillment of the Requirements for the
Degree of

Master of Science in
Aeronautics and Astronautics

at the

Massachusetts Institute of Technology

September 1992

Signature of Author_____
Department of Aeronautics and Astronautics
August 19, 1992

Approved by_____
John J. Turkovich
Technical Supervisor, Charles Stark Draper Laboratory

Certified by_____
Professor Joseph F. Shea
Thesis Supervisor

Accepted by_____
Professor Harold Y. Wachman
Chairman, Department Graduate Committee

AN AUTOMATED TOOL FOR FORMULATING LINEAR COVARIANCE ANALYSIS PROBLEMS

by

Andrew W. Lewin

Submitted to the Department of Aeronautics and Astronautics
on August 21, 1992 in partial fulfillment of the requirements
for the Degree of Master of Science in Aeronautics and Astronautics

ABSTRACT

This thesis describes the design and development of an automated tool for developing linear covariance analysis simulations. A generic specification language was developed for navigation linear covariance analysis problems. This language served as the basis for the user interface of the automated tool. The ability to convert linear covariance analysis specifications to Ada code was demonstrated with the aid of ASTER, a software development tool that generates source code from functional specifications.

Although work is ongoing, the linear covariance analysis tool has demonstrated that using a design language as the basis of the computer code can save valuable development time, while increasing reliability and maintainability of the resulting code. Automation also allows the engineer to take advantage of inherent properties of the analysis equations to improve computational efficiency.

Thesis Supervisor: Mr. John J. Turkovich
Title: Staff Engineer, The Charles Stark Draper Laboratory, Inc.

Thesis Supervisor: Professor Joseph F. Shea
Title: Adjunct Professor of Aeronautics and Astronautics

Acknowledgements

Many people at Draper Lab have made my stay a very enjoyable one. I want to offer my thanks to Milt Adams for his stimulating conversation and interest in my work despite his many duties. I also thank Eva Moody for her assistance in everyday survival at the Lab. She makes everyone's job much easier. Thanks also to Professor Shea, who took me on and became an insightful advisor with many excellent suggestions, and to Ken Spratlin and Stan Shepperd who taught me linear covariance analysis and who provided me much guidance in the design of the automated tool.

I appreciate the long hours put in on my behalf by Dave Chamberland and Rick Brenner. I offer a special thanks to Dave for not killing me, despite the fact that I was consistently interrupting his work with bug reports or requests for new features. I must offer my greatest thanks to John Turkovich, who brought me to the Lab three years ago and has given me exciting projects and a nurturing environment ever since.

I thank my parents for helping me to get to this point, and for dealing with all the difficulties it caused. Most of all, I want to thank my fiancee Ellen, whose love and support enabled me to successfully complete this thesis.

This thesis was prepared at the Charles Stark Draper Laboratory by company sponsored research (CSR-103), internal research and development (IR&D 462), and the MIT Space Grant Program.

Table of Contents

Chapter 1: Introduction

Improvements in computer hardware over the last decade have produced orders of magnitude improvements in both speed and memory capabilities. These advances have led to revolutionary changes in software as well. Software began with machine languages which provided only the most basic commands. Nearly trivial processes required an engineer to write many lines of code. As a result, assembly languages were developed. These were somewhat easier to use than machine language, and the use of words or abbreviations of words was certainly helpful. However, assembly language commands were still prohibitively primitive. Therefore, higher level languages were developed that were much easier to use. The commands were actual english words, and the command structure resembled sentences.

At the same time, software systems have been growing exponentially in size and complexity. Advances in microprocessor speed and memory capabilities have lead system designers to ever increasing demand for software functionality. The Apollo 11 astronauts successfully landed on the moon guided by a computer with only 38,000 words of memory. Now, 23 years later, many aerospace software systems require several millions of lines of code.

The purpose for creating high-level programming tools is to reduce the time necessary to design, build, and test, and maintain a software system. As software systems become increasingly complex, design and coding capabilities must evolve accordingly. These gains are accomplished at the expense of memory utilization, execution speed, and number of lines of machine-language code. However, the lower efficiencies are far outweighed by the cost and schedule benefits of the reduced development time.

Research and commercial efforts are now developing the next revolution in software development technology. Systems are being developed that raise software specification to design specifications rather than computer code. This new generation of compiler offers substantial improvements in development time for computer systems in three ways.

First, the development time is dramatically reduced. The engineer does not need to spend time either writing code himself or describing the design to a computer programmer who probably does not understand the design language the engineer naturally uses to describe the design or the engineering aspects of the design.

Second, the software development process is made more reliable. An automated system, once matured, generates code nearly error-free. Although the use of automated software generators does not eliminate the need for testing, it can greatly decrease the number of errors present in the system, and thereby reduce the amount of time spent debugging. Furthermore, enabling an engineer to specify the design in familiar terms rather than the sometimes complex workings of a programming language reduces the amount of work required by the engineer, which reduces the opportunity for errors to be introduced into the system.

Third, automating the software development process makes the software system much easier to maintain. Early systems were small enough that maintenance was of little concern. However, as software systems have become more complex, maintenance has become very expensive. By specifying a design in terms of a design language rather than computer code, changes can be made much more easily. The person responsible for making the change does not have to decipher the code. Rather, he or she just has to look at the design and make the necessary modifications.

One application which can particularly benefit from automation is linear covariance analysis. The purpose of linear covariance analysis is to evaluate the performance of dynamic systems. One common application of linear covariance analysis is error modelling for guidance and navigation systems. Guidance and navigation accuracy is a major design concern for both military and civilian applications. Linear covariance analysis enables an engineer to evaluate how a particular set of instruments and navigation aids affect the accumulation of error throughout a mission. Therefore, linear covariance analysis plays a critical role in the design tradeoff between additional or higher performance instruments and

the additional cost associated with these items.

This thesis details the development of a linear covariance analysis software development tool. Chapter 2 describes the linear covariance analysis algorithm and its application to navigation systems. Chapter 3 describes the development of a design language for linear covariance analysis problems. This process was necessary because the navigation system analysis community has not yet developed a design language standard. Chapter 4 describes the details of the automated linear covariance analysis tool that was developed. Chapter 5 evaluates the system that was developed, and Chapter 6 identifies desired improvements to the tool to extend its functionality and better serve its users.

Chapter 2: Linear Covariance Analysis

Linear covariance analysis is a technique for analyzing error performance of dynamic systems. The error is described relative to given nominal performance. The linear covariance analysis begins by describing the elements that comprise a mission. These elements are then combined to create a particular mission. The linear covariance analysis then performs a simulation of the specified mission and determines how errors propagate through the system during the course of the mission.

2.1 Linear Covariance Analysis Elements

Analysis of a mission must consider three basic elements: celestial bodies, platforms, and sensors. A celestial body supports ground-based beacons and determines the equations of motion of orbiters and maneuvering vehicles. Celestial bodies include the sun, planets, and moons. Platforms are bodies which carry sensors. Platforms include navigational aids such as beacons and satellites as well as maneuvering bodies such as rovers and vehicles. Finally, sensors are the instruments placed on board the platforms which provide navigation information through external measurements.

The core of any design process is modelling the physical process taking place. For linear covariance analysis, this requires the development of appropriate models for the celestial bodies, platforms, and sensors that comprise the mission to be evaluated. In order to design an automated tool for building linear covariance analysis simulations, it is necessary to determine which physical properties must be modelled. This involves identifying the relevant properties of the building blocks of a mission.

2.1.1 Celestial Bodies

Celestial bodies form the basis of any mission, since the purpose of the mission is to guide and navigate vehicles about these objects. The linear covariance analysis algorithm requires information about the bodies that determines how they interact with the vehicles and navigational aides involved in the mission. Obviously, the mass is a critical parameter, since it governs the gravitational motion of bodies in the vicinity of the body. However, the distribution of the mass also plays an important role in determining the motion of orbiting satellites, both natural and man-made. Other important quantities include the radius and atmosphere height, which determine occultations of an orbiting platform by the body. For problems involving multiple celestial bodies, the orbital elements of the body are required in order to evaluate the relative position and velocity of the bodies. Similarly, the rotation rate and rotation axis must be known to calculate the orientation of the celestial body relative to inertial space. This rotation is especially important for determining the location of surface-based platforms such as beacons and rovers.

2.1.2 Platforms

By definition, a platform is any entity which can carry sensors. For typical aerospace linear covariance analysis problems, there are four main platform types: vehicles, rovers, orbiters, and beacons. A maneuvering vehicle is an object capable of powered flight, and its motion is given by a predefined trajectory. It is described only by its trajectory. A rover is also capable of independent movement, but it is constrained to traverse a planetary surface. Like the maneuvering vehicle, a rover is also modelled by a trajectory. An orbiter moves under gravitational influences only with no powered maneuvers beyond simple stationkeeping. The motion of an orbiter is described by its orbital elements. Finally, beacons are navigational aids which are placed on the surface of a celestial body and have no means of independent locomotion. In addition to its location, a beacon model includes an elevation angle above the horizon requirement for communication with orbiting platforms. This parameter models surface irregularities or atmospheric effects such as refraction.

2.1.3 Sensors

Since the primary purpose of linear covariance analysis is to predict error performance of a guidance or navigation system, it is not surprising that the sensor models are the most

complicated. Instrument or sensor modelling involves the specification of four primary elements: error terms, measurement sensitivity, measurement variance, and constraints.

The sensor error terms provide a mathematical model of the physical characteristics of an instrument. Error term dynamics are expressed as differential equations. The terms of the differential equations are functions of the modelled errors present in the system as well as position and velocity errors of the host platform. However, linear covariance analysis requires that the sensor error dynamics be expressed as a series of linear first-order differential equations. Therefore, non-linear terms must be linearized or ignored, and higher-order differential equations must be reduced to a system of linear first-order differential equations.

Although the error terms can provide a very accurate model of the sensor performance, no model can ever be perfect. As a result, each differential equation usually includes a zero mean white noise error term. The purpose of this term is to capture the effects of known modelling deficiencies as well as unknown error sources, thereby preventing the filter from becoming overly optimistic.

The second sensor modelling parameter is the measurement sensitivity, **b**. The measurement sensitivity describes how the measurement would vary as a function of small changes in the elements of the state vector. Mathematically,

$$\mathbf{b} = \left(\frac{\delta q}{\delta \mathbf{e}}\right)^T$$

where q is the sensor measurement (usually scalar) and **e** is the full error state vector.

Measurement variance, the third sensor parameter, is used to identify the accuracy of a sensor measurement. The measurement variance is expressed as a single equation. In general, this equation is comprised of two parts. The first part is a sensor noise term. The noise term is used to model the accuracy of the instrument itself. However, this quantity may then be modified by a term relating to the conditions of the measurement to get the measurement sensitivity. For example, a sensor measurement may degrade with the measurement range or perhaps even the temperature of the sensor. By combining these two factors, an equation describing the accuracy of the measurement, the measurement variance, is determined.

Finally, constraints are used to identify whether a measurement can actually be performed. One common constraint for sensors is that they be able to "see" the target sensor. For example, radio communication is impossible through a planet, so a two-way range measurement using radio waves requires a line of sight constraint. Another constraint is a maximum operating range. If two sensors are too far apart, they may not have enough power to successfully take a measurement.

2.2 Mission Specification Procedure

Once the celestial body, sensor, and platform characteristics have been determined, the specification of a linear covariance analysis problem is broken down into four major steps. First, the celestial bodies, platforms, and sensors in the problem are specified. The next step is to select the measurements to be taken and their operating characteristics. Third, the simulation parameters, initial conditions, and desired outputs are determined. Finally, the output is evaluated and the process is repeated.

2.2.1. Specifying the Mission

The first step of linear covariance analysis is to specify the mission to be undertaken. This involves selecting the celestial bodies in the simulation. This may include sun, planets, or moons that affect motion or bodies such as stars which are used for navigational purposes. Then, the the vehicles, orbiters, rovers, and navigational aids are specified. For vehicles and rovers this involves providing the trajectory. Orbiters must be given an orbit about one of the bodies, and navigational aids such as beacons are placed at a fixed location on a planet or moon.

Once the platforms have been identified, the instruments aboard each platform are specified. First, the engineer selects the sensors to be placed on board each platform. Then, he or she specifies the modelling fidelity for each sensor. For example, an sensor specification may include as many as 50 or 100 error terms. However, they may not all be necessary, so the engineer specifies which error terms to use. The engineer can also declare an error term to be a consider state. Consider states are included in the covariance matrix, but they do not benefit from measurement update information. This is the equivalent of examining an error term which is not included in the actual filter. The state is not being estimated; it is only propagated.

2.2.2 Specifying the Measurements

Once the mission has been specified, the measurements taken by each instrument are determined. Sensors such as an inertial measurement unit (IMU) do not require this step because the measurement being taken does not depend upon other platforms in the mission. Placing the IMU on a platform is sufficient to determine what measurement to take. However, most instruments involve objects other than the host platform when taking a measurement. For example, an altimeter measures the distance from the surface of a celestial body. For the altimeter, specifying the measurement to be taken means identifying the celestial body to which the distance is measured.

In addition to determining the measurements to be taken, an engineer must decide how frequently the measurements are taken. Measurement frequency plays an important role in navigational errors. When more measurements are taken, the errors will naturally be reduced; however, each additional measurement adds to the amount of throughput required of the avionics system. Furthermore, each additional measurement has diminishing returns. Also, sensors may be turned on or off at specific times during a mission. On the Apollo missions, unused sensors had to be turned off because they overloaded the tiny flight computer. Another use of this feature is to model fault tolerance of the navigation system. By turning off sensors at specific times, an engineer can simulate the failure of that sensor.

2.2.3 Simulation Parameters, Initial Conditions, and Output Modes

Once the mission has been determined, several parameters involved in the linear covariance analysis must be specified. First, the initial covariance matrix must be given to provide an initial condition for the integration and propagation of the error covariance. Since the covariance matrix is often very large, this can be a time-consuming process; however, the covariance matrix is symmetric.

Another useful simulation parameter is measurement underweighting. Measurement underweighting provides a mechanism for not taking full advantage of the information provided by a measurement. This is particularly useful when a reduced order system is being used or in a region where the system is particularly non-linear. If the measurement underweighting is not used, the algorithm can become overly optimistic.

The final step of the mission specification process involves defining parameters that affect the simulation. These include the starting and ending times as well as the time step. The time step plays an important role in the quality of the simulation. A small time step yields an accurate simulation, but it requires many calculations which can cause the analysis process to take a long time to perform. For some missions, it may be desirable to change the time step during the mission. The time step is chosen to be very small during phases where the dynamics are undergoing rapid change and longer during relatively quiet periods.

2.2.4. Output Evaluation

Once the mission has been defined and the linear covariance simulation has been performed, the true analysis begins. Linear covariance analysis is simply a tool that enables an engineer to simulate the error propagation of a navigation system. With the data gleaned from the results of the linear covariance analysis algorithm, an engineer must determine whether or not the current mission scenario satisfies the given requirements. If not, he or she must examine the data to determine what caused the mission to not meet the requirements. Then the engineer uses this information to modify the mission and repeat the entire process. In order to make this process effective, the engineer must have a variety of ways to examine the output of the linear covariance analysis. Selecting the desired outputs and the method of viewing them constitutes the final step in the linear covariance analysis process.

2.3. Performing Linear Covariance Analysis

The purpose of linear covariance analysis is to provide a model for determining the error performance of a navigation system. The basis of the linear covariance analysis algorithm is the assumption that instrument errors and vehicle motion can be modelled as a set of linear differential equations driven by zero mean white noise:

$$\dot{\mathbf{e}} = \mathbf{F}\mathbf{e} + \mathbf{G}\mathbf{u} \tag{1}$$

where **e** is the vector of state errors and **u** is a vector of white noise. The matrix **F**, or state dynamics matrix, describes the linearized dynamics of the platforms and sensors involved in the analysis. The matrix **G**, or noise distribution matrix, models the impact of the noise on each of the error terms. If the noise sources are independent for each platform and

sensor error term, G will be diagonal or block diagonal, depending upon how the noise affects vector-valued errors. In most cases, G will be an identity matrix.

Linear covariance analysis is not a deterministic simulation; rather, it is a *statistical* simulation. The primary output of the algorithm is the covariance matrix. The covariance matrix represents the state error variances and covariances, not the errors for an actual mission. Mathematically, the covariance matrix is given by the equation:

$$\mathbf{E} = \text{expected-value}(\mathbf{e}\,\mathbf{e}^{\mathrm{T}}) = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \cdots \\ \sigma_{xy} & \sigma_y^2 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

The diagonal terms of the covariance matrix are called the variances. The square root of the variance is the standard deviation. The standard deviation is a useful measure because it has a simple physical interpretation. For an actual mission, the error can be expected to be less than the standard deviation 68.27% of the time, less than double the standard deviation 95% of the time, and less than three standard deviations 99.73% of the time.

The off diagonal terms determine the correlation between two states. The correlation coefficient is given by:

$$\zeta_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y} \qquad (-1 \leq \zeta_{xy} \leq 1)$$

$\zeta_{xy} = 0$ means that states x and y are not statistically correlated, whereas $\zeta_{xy} = \pm 1$ means that they are 100% correlated.

The linear covariance analysis algorithm consists of two primary tasks: 1) propagating the covariance matrix and the position and velocities of the platforms and celestial bodies in the mission, and 2) updating the covariance matrix when measurements are taken. Before this process can begin, the covariance matrix and state vector must be initialized. These quantities represent the cumulative effects of everything that has happened prior to initiation of the simulation. Therefore, the linear covariance algorithm describes the accumulated error performance for all time up to the end of the simulation, not just the short window actually simulated.

2.3.1 Propagating the Covariance Matrix and the Platform and Celestial Body State Vectors

Once the simulation begins, the first major step is to update the error covariance matrix and the platform states. For a linear time-varying differential equation driven by white noise such as equation (1), the covariance matrix dynamics are given by the Lyapunov differential equation:

$$\dot{\mathbf{E}} = \mathbf{FE} + \mathbf{EF}^{\mathbf{T}} + \mathbf{Q} \tag{2}$$

where **E** is the covariance matrix, **F** captures the dynamics of the error state vector **e**, and **Q** is the noise intensity or process noise matrix. The process noise matrix is the product of the noise distribution matrix **G** and the noise vector **u**.

The method of determining platform states depends upon the type of platform. Beacons are fixed to a planet, so their position and velocity can be determined from the translation and rotation of the planet. Orbiters are incapable of self-propulsion, so their motion is calculated by integrating the gravitational acceleration vector to get the velocity, and the velocity vector to get the position. Rovers and maneuvering vehicles, on the other hand, are capable of providing forces other than gravitational attraction. Therefore, the position and velocity vectors for these platforms as a function of time are given as inputs to the algorithm.

When a mission specification contains more than one celestial body, all celestial body position and velocity vectors must also be maintained. This is accomplished by simply integrating their state vectors in the same way as for orbiters. This process is unnecessary when only one celestial body is included in a mission because that body is used as the inertial center of the computational system.

Accuracy is extremely important for the systems being evaluated by the linear covariance analysis algorithm. Therefore, accuracy of the simulation is equally important. Experience has shown that a 4^{th}-order Runge-Kutta integration scheme is sufficiently accurate for all state and covariance integrations. During periods such as vehicle coasting, where changes are not happening as rapidly, lower-order integration schemes may also be acceptable.

2.3.2 Measurement Updating

The second step is to process all the measurements. For simplicity, the updating has been restricted to scalar updates, in other words, the measurements are processed sequentially, even when several measurements have been scheduled for the same time. The measurements still occur at the correct time step, but they use a covariance matrix which has already been updated by previously processed measurements from the same time step.

To begin, the algorithm determines whether a measurement is scheduled to take place. If so, the algorithm must then ensure that the sensor is on and all operating constraints are met. If all of these conditions are met, a measurement is performed. For each measurement, the covariance matrix is updated to reflect the new information. This is accomplished by the following series of equations:

First, the temporary vector **U** and the measurement residual variance, $\sigma^2_{residual}$ are calculated:

$$\mathbf{U} = \mathbf{E}\,\mathbf{b}$$
$$\sigma^2_{residual} = \mathbf{b}^T\,\mathbf{U}$$

The residual is a scalar measure of the states with respect to the scalar measurement. If the residual is small, the states are well-known. Then, the measurement weighting vector is formed:

$$\hat{\mathbf{w}} = \left(\frac{1}{(1+\beta_{uw})\,\sigma^2_{residual} + \alpha^2_{meas}}\right)\mathbf{U}$$

where β_{uw} is the measurement underweighting and α^2_{meas} is the measurement variance.

The weighting vector identifies the relative importance of the various error terms in updating the covariance matrix. If an error term is only a consider state, it is not used to update the state vector, so the appropriate element of the weighting vector is set to zero. The array **C** identifies whether each error term is a consider state.

If **C**(i) is true, $\hat{\mathbf{w}}$(i) is set to 0

The update proceeds with the equation:

$$\mathbf{w} = \frac{\mathbf{U}}{\sigma^2_{residual} + \alpha^2_{meas}}$$

If consider states or underweighting are used, $\mathbf{w}$ and $\hat{\mathbf{w}}$ will not be in the same direction. For optimal filters, (no underweighting or consider states) the covariance matrix is updated using the equation:

$$\mathbf{E}_{new} = \left(\mathbf{I} - \mathbf{w}\mathbf{b}^T\right)\mathbf{E}$$

For sub-optimal filters (those with underweighting and/or consider states), the Joseph form is used:

$$\mathbf{E}_{new} = \left(\mathbf{I} - \hat{\mathbf{w}}\mathbf{b}^T\right)\mathbf{E}\left(\mathbf{I} - \hat{\mathbf{w}}\mathbf{b}^T\right)^T + \alpha^2_{meas}\hat{\mathbf{w}}\hat{\mathbf{w}}^T$$

Here, it can be seen that by introducing consider states or underweighting, $\hat{\mathbf{w}}$ is smaller than it otherwise would be, so $\mathbf{I} - \mathbf{w}\mathbf{b}^T$ is larger, which makes $\mathbf{E}$ larger, indicating that the vehicle has more uncertainty than it would without the sub-optimalities. Finally, the symmetry of the covariance matrix is retained through the equation

$$\mathbf{E} = \frac{\mathbf{E} + \mathbf{E}^T}{2}$$

Without this equation, the covariance matrix can lose symmetry due to computer round-off errors.

2.4 Applying Linear Covariance Analysis to Navigation Problems

Although the terminology used to describe the linear covariance algorithm in Section 2.3 is specific to navigational linear covariance analysis problems, the technique is not. The linear covariance analysis algorithm must be applied to the objects described in Section 2-1.

The first part of linear covariance analysis is propagating the covariance matrix and state vectors. The covariance matrix propagation involves two inputs, the dynamics matrix $\mathbf{F}$ and the noise intensity matrix $\mathbf{Q}$. The dynamics matrix is created by creating a system of linear equations that describe the platform state error dynamics and the sensor error terms. Celestial body state error dynamics are not modelled. These linear equations contains terms

based on error terms in the linear covariance analysis as well as the white noise associated with each of the sensors. The terms of the differential equations based upon current errors are expressed in matrix form. This matrix is the dynamics matrix **F**. The noise terms are also converted to a matrix form, creating the noise intensity matrix **Q**.

Some of the differential equations include terms which are a function of the position or velocity of platforms or celestial bodies in the mission. Therefore, these state vectors must be propagated in time. The propagation is defined by simple integration of acceleration to get velocity and velocity to get position. The propagation requires an initial condition to be provided for both the position and velocity of each object.

The second step of the linear covariance analysis is to perform measurements and update the covariance matrix to account for the new information provided by those measurements. Before this can occur, the algorithm must determine which measurements should be attempted; then, the update occurs for any successful measurements. This process involves three checks: the sensor must be on, a measurement must be scheduled, and the environment must meet all sensor constraints. Once a mission is specified, the measurements to be performed must also be identified. This includes the data involving when a sensor in on or off and how often the sensor measurements will be taken. The constraints, on the other hand, are a property of the sensor.

When a measurement is performed, three inputs are required: the measurement sensitivity vector **b**, underweighting factor β_{uw}, and the consider state array. Each sensor has a measurement sensitivity specification. This specification includes values for terms in the error state vector that relate to the sensor itself, state errors of the platform it resides upon, and state errors of any other platforms involved in the measurement. These specifications are used to define the appropriate elements of the measurement sensitivity vector. The remaining elements are set to zero. The measurement underweighting factor is used to prevent the linear covariance analysis from becoming too optimistic in its predictions of error performance. It is a global constant used in all calculations.

Finally, the algorithm allows an engineer to select consider states, which are error terms that are propagated, but the information captured by those error terms is not used by the vehicle to improve its estimation of the vehicle states. These terms are identified as part of describing the desired modelling fidelity of a sensor on board a platform.

2.5 Improving the Computational Performance of the Algorithm

Since a sensor can be required to perform many measurements during a time step, the update process may be repeated several times for a sensor in the same time step. An example of this situation is a satellite constellation containing 20 orbiters, each carrying a range measurement device. This instrument is commanded to estimate the range from the host orbiter to all the other satellites in the constellation. Therefore, on the time step when a measurement is scheduled to occur, the algorithm checks to see which of the other 19 satellites meet the constraints for the range measurement. In this case, the constraint would probably be that the central planet is not between the two orbiters. For each measurement which is actually taken, the covariance matrix must be updated. Therefore, the covariance matrix update equations could be evaluated as many as 19 times in one time step for just one sensor. If all orbiters were scheduled to take measurements at the same time, as many as 380 updates could be required.

As this example illustrates, linear covariance analysis problems can be computationally intensive. Therefore, it is worthwhile to examine whether an automated tool can be used to reduce the computation. This study determined that several inherent properties of the linear covariance problem can be exploited to dramatically reduce the number of calculations required on each iteration.

2.5.1 Computational Properties of the Linear Covariance Analysis Algorithm

Before any progress can be made, the sources of the calculation time must be identified. The majority of the calculations occur in the Runge-Kutta integration and covariance matrix update for each measurement. The calculations involved in these two processes are evaluated for an error state vector containing n elements.

Runge-Kutta Integration of Lyapunov differential equation $\dot{\mathbf{E}} = \mathbf{FE} + \mathbf{EF}^T + \mathbf{Q}$

The Runge-Kutta integration algorithm is as follows:

Given:	Initial covariance matrix	$\mathbf{E}_0$
	Noise matrix	$\mathbf{Q}$
	Vector	$\mathbf{k} = (0.0 \;\; 0.5 \;\; 0.5 \;\; 1.0)$

loop from i=1 to 4

if i=1,

let $\mathbf{E} = \mathbf{E}_0$

otherwise,

let $\mathbf{E} = \mathbf{E}_0 + \mathbf{k}_i * \Delta\mathbf{E}_{i-1}$

let $\Delta\mathbf{E}_i = \Delta t\,(\mathbf{FE} + \mathbf{EF}^T + \mathbf{Q})$

end loop

$$\mathbf{E} = \mathbf{E}_0 + (\Delta\mathbf{E}_1 + 2*(\Delta\mathbf{E}_2 + \Delta\mathbf{E}_3) + \Delta\mathbf{E}_4)/6$$

$$\mathbf{E} = \frac{\mathbf{E} + \mathbf{E}^T}{2}$$

As given, this algorithm has the following computational properties:

Equation		Operations	Type
$\mathbf{E} = \mathbf{E}_0 + \mathbf{k}_i * \mathbf{E}_{i-1}$		n^2	multiplication
(executed 3 times)		n^2	addition
$\Delta\mathbf{E}_i = \Delta t\,(\mathbf{FE} + \mathbf{EF}^T + \mathbf{Q})$	**FE:**	n^3	multiplication
(executed 4 times)		$n^2(n-1)$	addition
	$\mathbf{EF}^T$:	n^3	multiplication
		$n^2(n-1)$	addition
	rest:	$2n^2$	addition
		n^2	multiplication
$\mathbf{E} = \mathbf{E}_0 + (\Delta\mathbf{E}_1 + 2*(\Delta\mathbf{E}_2 + \Delta\mathbf{E}_3) + \Delta\mathbf{E}_4)/6$		n^2	multiplication
		n^2	division
		$4n^2$	addition

Total:		
	$3(n^2) + 4(2n^3 + n^2) + n^2$	multiplication
	n^2	division
	$3(n^2) + 4(2n^3) + 4n^2$	addition
	$16n^3 + 16n^2$	floating-point operations

Measurement updates

Every time a measurement takes place, the covariance matrix must be updated to account for errors introduced by that measurement. The process for updating the covariance matrix has the following computational properties:

Equation	Operations	Type
$\mathbf{U} = \mathbf{E}\,\mathbf{b}$	n^2	multiplication
	$n(n-1)$	addition
$\sigma^2_{residual} = \mathbf{b}^T\,\mathbf{U}$	n	multiplication
	$n-1$	addition
$\hat{\mathbf{w}} = \left(\frac{1}{(1+\beta_{uw})\,\sigma^2_{residual} + \alpha^2_{meas}}\right)\mathbf{U}$	n	multiplication
$\mathbf{E}_{new} = (\mathbf{I} - \hat{\mathbf{w}}\mathbf{b}^T)\,\mathbf{E}\,(\mathbf{I} - \hat{\mathbf{w}}\mathbf{b}^T)^T + \alpha^2_{meas}\hat{\mathbf{w}}\hat{\mathbf{w}}^T$	$2n^3 + 2n^2$	multiplication
	$2n^3 - n^2$	addition
	$3n^2$	subtraction

Total:	$2n^3 + 3n^2 + 2n$	multiplication
	$2n^3 - 1$	addition
	$3n^2$	subtraction
	$4n^3 + 6n^2 + 2n - 1$	floating-point operations

At first glance, this computation may seem trivial compared the Runge-Kutta integration of the covariance matrix since the integration is $O(n^3)$ whereas with some manipulation the update equations are $O(n^2)$. However, this computation is repeated as many times as there are measurements. For some problems, the number of measurements can become a significant percentage of the number of elements in the error state vector, so the two calculations may be at the same order. Furthermore, improvements in the integration algorithm can reduce it to $O(n^2)$ as well.

2.5.2 Improving Computational Efficiency

By taking advantage of the physical properties of the problem being modeled by the linear covariance analysis, the amount of computation can be dramatically reduced. In particular, the dynamics matrix and measurement sensitivity vectors are almost always sparse. Furthermore, the location of these zeros is well defined. This knowledge is sufficient to allow changes in the algorithm to greatly decrease the required computation time.

Since the dynamics matrix captures the differential equations describing the propagation of errors, the formulation of the error state vector directly determines the format of the dynamics matrix. The error state vector is arranged by platform. The first terms in the vector for each platform are the position and velocity errors for that platform. These are followed by the error terms for each sensor on board the platform. Sensors such as inertial

measurement units (IMU) which affect the velocity error of the host platform and sensors whose error terms depend upon the position and velocity error of the host platform are placed first. This configuration is shown in Figure 2-1.

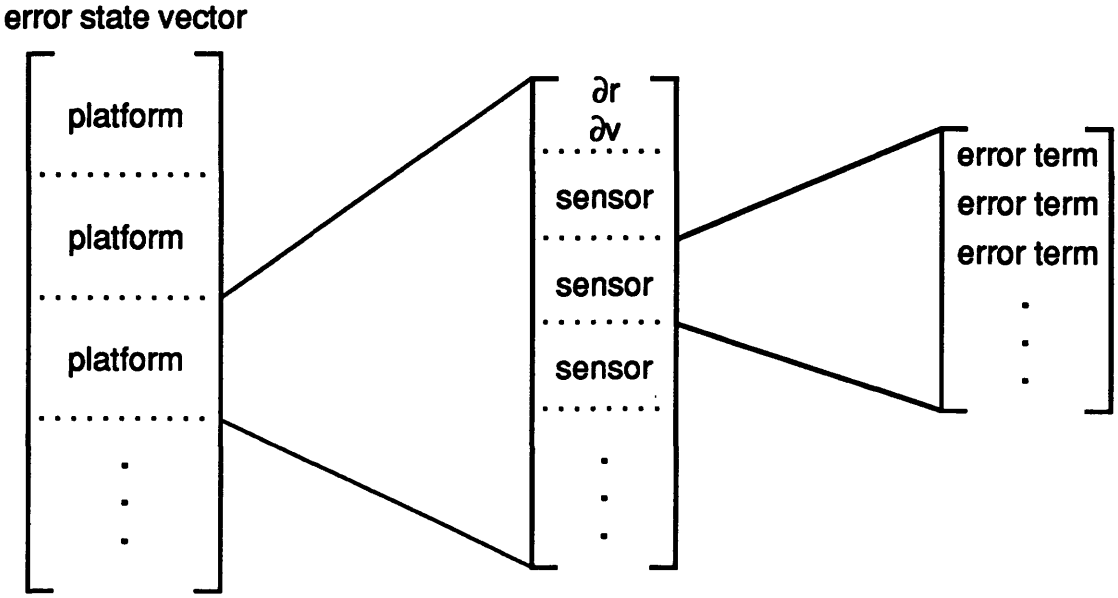


Figure 2-1: Formulation of the error state vector

This approach leads to a major advantage. The propagation of errors for most sensors are solely dependent upon errors in that sensor. Therefore, the dynamics for each sensor will appear in a block along the diagonal of the dynamics matrix. A few sensors either depend upon or affect the position and velocity errors of the host platform. These sensors have additional dynamics in the off-diagonal locations corresponding to the host platform position and velocity errors. By placing these sensors first, the dynamics for these sensors and the host platform position and velocity errors are contained in a larger block, while the remaining sensors have individual blocks along the main diagonal of the dynamics matrix. The form of the dynamics matrix is shown in Figure 2-2.

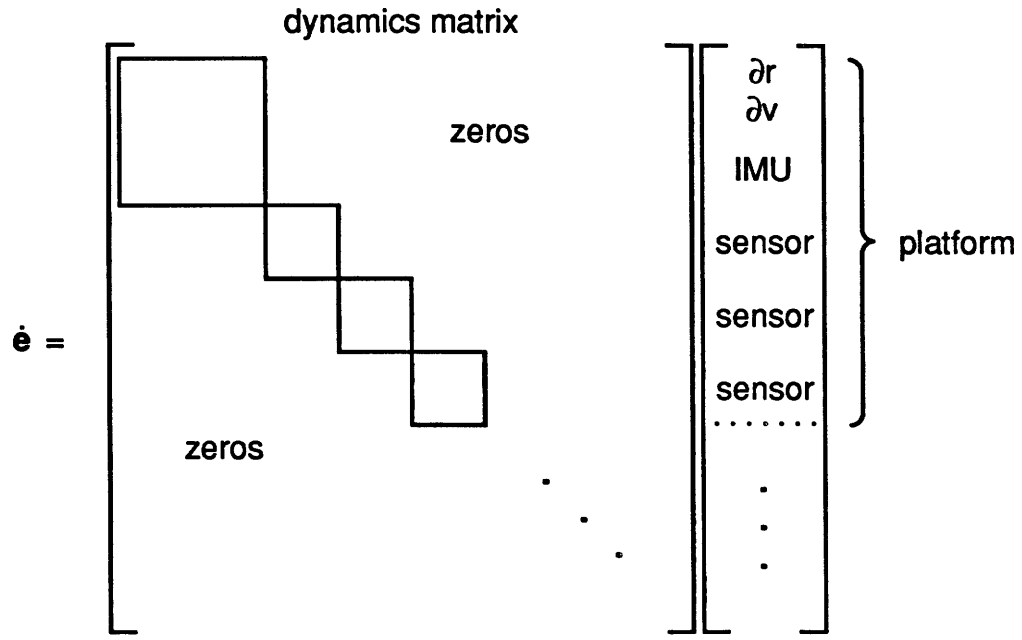


Figure 2-2: Dynamics matrix in block diagonal form.

With the dynamics matrix in this form, the multiplication of the dynamics matrix and the covariance matrix is much simpler. For example, if the n x n dynamics matrix is broken into m equal size matrix blocks along the main diagonal, the number of calculations is n^3/m multiplications and $n^2(n-m)/m$ additions. The full matrix case described earlier corresponds to m = 1, yielding n^3 multiplications and $n^2(n-1)$ additions. At the other extreme, a diagonal dynamics matrix, m = n, requires only n^2 multiplications and no additions.

In practice, the savings can approach the order of magnitude reduction seen with the diagonal matrix case. Many error terms are modeled as first-order Markov processes, so the error term dynamics are solely dependent upon the error term itself. Therefore, the only contribution to the dynamics matrix lies directly on the main diagonal. This enables the blocks associated with each matrix to be broken down even further to gain the maximum benefit.

The other major improvement occurs in the update equations. This improvement takes two forms, manipulating the update equation to reduce the order of the computations as well as the number of mathematical operations required. This begins with a modified version of the Joseph form developed by Stan Shepperd[1] :

$$\mathbf{E}_{new} = \mathbf{E} - k\mathbf{w}\mathbf{w}^T + k(\hat{\mathbf{w}} - \mathbf{w})(\hat{\mathbf{w}} - \mathbf{w})^T$$

where $k = \sigma^2_{residual} + \alpha^2_{meas}$

In this form, $2n^2$ multiplications and $2n^2$ additions are required as compared to the original $O(n^3)$ equation. This yields the following computational properties:

Equation	Operations	Type
$\mathbf{U} = \mathbf{E}\,\mathbf{b}$	n^2	multiplication
	$n(n-1)$	addition
$\sigma^2_{residual} = \mathbf{b}^T\mathbf{U}$	n	multiplication
	$n-1$	addition
$\hat{\mathbf{W}} = \left(\frac{1}{(1+\beta_{uw})\,\sigma^2_{residual} + \alpha^2_{meas}}\right)\mathbf{U}$	n	multiplication
$\mathbf{E}_{new} = \mathbf{E} - k\mathbf{w}\mathbf{w}^T + k(\hat{\mathbf{w}} - \mathbf{w})(\hat{\mathbf{w}} - \mathbf{w})^T$	$2n^2 + 2n$	multiplication
	n^2	addition
	n^2	subtraction

Total:		
	$3n^2 + 4n$	multiplication
	$2n^2 - 1$	addition
	n^2	subtraction
	$6n^2 + 4n - 1$	floating-point operations

Sheppard's form completely eliminated $4n^3$ computations at the expense of just 2n. It should also be noted that the order of operations can be very important. In the $k\mathbf{w}\mathbf{w}^T$ term, multiplying k and w and then multiplying $\mathbf{w}^T$ requires $n^2 + n$ calculations whereas multiplying $\mathbf{w}\mathbf{w}^T$ uses $2n^2$ calculations. This is because kw is a vector multiplication and $k(\mathbf{w}\mathbf{w}^T)$ is a matrix multiplication. A similar situation arises with $k(\hat{\mathbf{w}} - \mathbf{w})(\hat{\mathbf{w}} - \mathbf{w})^T$. The performance properties given above assume that the more efficient computation is used and that $\hat{\mathbf{w}} - \mathbf{w}$ is performed only once.

However, Sheppard's equation can be further reduced with some additional manipulations. To begin, replace w and $\hat{\mathbf{w}}$ with their definitions, yielding

$$\mathbf{E}_{new} = \mathbf{E} - \frac{1}{k}\mathbf{U}\mathbf{U}^T + k\left(\frac{1}{(1+\beta_{uw})\,\sigma^2_{residual} + \alpha^2_{meas}} - \frac{1}{k}\right)^2\mathbf{U}\mathbf{U}^T$$

This step yields an important restriction. This equation is only valid for a system with no consider states, because when a state is considered, it is zeroed in $\hat{\mathbf{w}}$ but not in w, so w and

$\hat{\mathbf{w}}$ are in different directions, which prevents the factoring out of $\mathbf{U}\mathbf{U}^T$. Most linear covariance problems do include consider states; however, pure underweighting is common in filtering problems.

Next, multiply k into the expression in parenthesis and combine the $\mathbf{U}\mathbf{U}^T$ coefficients:

$$\mathbf{E}_{new} = \mathbf{E} + \left[\left(\frac{k}{(1 + \beta_{uw})\,\sigma^2_{residual} + \alpha^2_{meas}} - 1\right)^2 - 1\right]\frac{1}{k}\mathbf{U}\mathbf{U}^T$$

In this form, both the multiplications and additions are cut in half. Thus, in final form, the update equations have the following computational properties (ignoring zero order calculations):

Equation	Operations	Type
$\mathbf{U} = \mathbf{E}\,\mathbf{b}$	n^2	multiplication
	$n(n-1)$	addition
$\sigma^2_{residual} = \mathbf{b}^T\,\mathbf{U}$	n	multiplication
	$n-1$	addition
$k = \sigma^2_{residual} + \alpha^2_{meas}$	1	addition
$\mathbf{E}_{new} = \mathbf{E} + \left[\left(\frac{k}{(1 + \beta_{uw})\,\sigma^2_{residual} + \alpha^2_{meas}} - 1\right)^2 - 1\right]\frac{1}{k}\mathbf{U}\mathbf{U}^T$		
	n^2	multiplication
	n^2	addition

Total:		
	$2n^2 + n$	multiplication
	$2n^2$	addition
	$4n^2 + n$	floating-point operations

This is down from $4n^3 + 6n^2 + 2n$ floating point operations required for the original form of the measurement update equations, and $6n^2 + 4n - 1$ from using Shepperd's form of the Joseph update equation. However, this set of equations is only valid for simulations without consider states. Nonetheless, this form has added elegance because it can be reduced to the same form as the optimal filter. First, define the constant κ such that

$$\mathbf{E}_{new} = \mathbf{E} - \kappa\frac{1}{k}\mathbf{U}\mathbf{U}^T$$

Then, replace $\mathbf{U}$ with $\mathbf{E}\mathbf{b}^T$ and $\frac{1}{k}\mathbf{U}$ with $\mathbf{w}$ so

$$\mathbf{E}_{new} = \mathbf{E} - \kappa\mathbf{w}\mathbf{b}^T\mathbf{E}$$

Factoring out E, this gives the same functional form as the optimal case with the addition of the constant κ:

$$\mathbf{E}_{new} = (\mathbf{I} - \kappa \mathbf{w}\mathbf{b}^T)\mathbf{E}$$

Additional reductions in the number of computations can be obtained because of zeros in the measurement sensitivity vector. When a measurement is taken, the only non-zero elements in the measurement sensitivity vector are the ones corresponding to the sensor taking the measurement, any sensors it is communicating with, and the position and velocity errors associated with the platforms housing each sensor. When multiplying the measurement sensitivity vector and the covariance matrix, the columns that would be multiplied by the zeros can be ignored. Therefore, if m out of n elements of the measurement sensitivity vector are non-zero, only m^2 multiplications and m(m-1) additions are necessary as compared to n^2 multiplications and n(n-1) additions for the full matrix case.

When the full measurement sensitivity vector is used to update the covariance matrix, the number of calculations rises dramatically as new measurements are added. This is because each new sensor increases the number of calculations in two ways. First, the sensor increases the number of measurements, which raises the number of times the update equations must be performed by an equal amount. Second, the additional sensor increases the size of the covariance matrix, which means that more computations must take place for each update cycle for all previously defined measurements as well as the new ones. By taking advantage of the zeros, the number of calculations associated with a single measurement is invariant with the total size of the covariance matrix. Therefore, the only additional computation required is the extra update of the covariance matrix.

One other improvement is possible, although it does not offer the same magnitude of improvement as the other options. The covariance matrix is symmetric, so only a triangular matrix including the main diagonal must be calculated. This can reduce both the integration and update calculations by nearly a factor of two and also potentially save an equal amount of memory, but implementing a series of calculations to operate only on the upper or lower triangular portion of the matrix is relatively complicated as compared to the gain. However, if computational performance is a problem, this feature may be exploited.

Chapter 3: Linear Covariance Analysis Natural Design Language

A key element of the design process is peer evaluation, especially in large projects. Thus, an engineer must be able to effectively communicate his or her design to other engineers. A good design language provides a common format for design data that helps to ensure clarity and reliability in this communication. The design specification consists of the essential elements of the design; no additional information is introduced through the specification language.

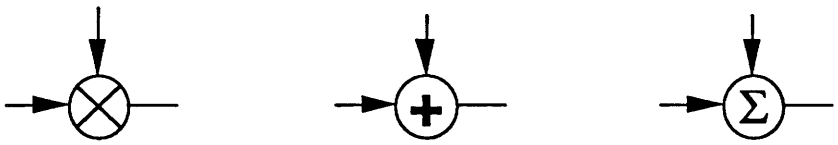
A good specification language gets at the heart of how an engineer thinks about the problem at hand. The specification language is then designed to mimic this thought process. Once again, extraneous information must be ignored, leaving only the essence of the design. The problem with developing a design language for linear covariance analysis is that the only analysis tool currently available is computer code. In fact, many linear covariance analysis algorithms are generated by putting together old pieces of code and making modifications as necessary. This approach obscures the true application design with issues related to the implementation of the design on a computer such as the language, compiler, computation environment, and similar concerns that have nothing to do with linear covariance analysis. Worse, over time, these computer related issues become a part of the design process of the engineer developing a linear covariance analysis problem.

In order for a design specification technique to be successful, this cycle must be broken. This can be accomplished by having an application engineer step back and think about the elements of the linear covariance analysis problem and how these elements fit together to form a mission specification. Many years of experience with thinking about the problem

one way are not easily replaced with a different one. The engineer must be frequently reminded to avoid any concerns about the implementation of the algorithm on a computer. In this way, the pertinent design information can be extracted from computer-related design issues and an application specification language can be developed.

The purpose of the design language is to capture the information necessary to perform the linear covariance analysis. Therefore, any language which accomplishes this task efficiently is acceptable. However, a good design language captures how an engineer thinks about the problem as well as the design data. This narrows the specification options significantly; on the other hand, it also implies that the optimum design language will be different for each individual. However, the differences between individual tastes are usually minor and do not pose a major threat to the development of a good design language.

Many languages also allow room for minor modifications to meet these differences. For example, in control block diagrams, the symbols



all represent addition. Although a design language may select any one of these three symbols, the basic concept of block diagram connectivity and functionality is unaffected.

This chapter provides a design language for linear covariance analysis, with the understanding that small changes may eventually be desirable until the engineering community as a whole decides upon a common specification language.

3.1 Linear Covariance Analysis Elements

Linear covariance analysis missions are comprised of three primary elements: celestial bodies, platforms, and sensors. Celestial bodies are the stars, planets, moons, and other gravitational bodies. Platforms are vehicles, rovers, orbiters, and beacons which carry sensors. Sensors are the instruments which perform measurements and provide information about the environment of the platforms they reside upon.

Celestial bodies

The only design information for a celestial body is a set of modelling parameters used to describe the body. For this purpose, a simple list of the parameters and their values is sufficient. A sample celestial body is shown.

Body: Earth

Physical properties:

Gravity constant (μ):	$3.986*10^{14}$ m^3/s^2
Radius:	6378 km
Atmosphere height:	25 km
Rotation rate:	$7.2921*10^{-5}$ rad/s

Orbital properties:

Central body:	Sun
Semi-major axis:	1.0 a.u.
Eccentricity:	0.01
Inclination:	8°
Longitude of the ascending node:	0°
Argument of pericenter:	0°
True anomaly:	147°

Platforms

The properties of the different platforms are defined within the automated system, so no specification language is necessary. The development of a design language must consider the general properties of a platform. Some design parameters include whether the platform must rest on a surface, whether the platform is capable of self-propulsion, what external forces act on the body, and so forth.

Sensors

A sensor specification consists of four major parts, error terms, constraints, and measurement sensitivity, and measurement variance. The first step is to define the error terms that model the sensor performance. This is done by simply listing the system of linear first-order differential equations. A list of the variables and their physical significance is provided as well, although the analysis could proceed without it. The following specification is a simple model for an inertial measurement unit.

$$\dot{e}_R = e_V$$

$$\dot{e}_V = G_R e_R + u_V + M_x(S)e_\psi + e_{bias}^{accel} + M_{diag}(S)e_{sf}^{accel}$$

$$\dot{e}_\psi^{gyro} = M_{I\text{-}B}M_{B\text{-}NB}\left(e_{\dot{\psi}}^{gyro} + u_\psi^{gyro}\right)$$

$$\dot{e}_{\dot{\psi}}^{gyro} = \left(-\frac{1}{\tau_{\dot{\psi}}^{gyro}}\right)e_{\dot{\psi}}^{gyro} + u_{\dot{\psi}}^{gyro}$$

$$\dot{e}_{bias}^{accel} = \left(-\frac{1}{\tau_{bias}^{accel}}\right)e_{bias}^{accel} + u_{bias}^{accel}$$

$$\dot{e}_{sf}^{accel} = \left(-\frac{1}{\tau_{sf}^{accel}}\right)e_{sf}^{accel} + u_{sf}^{accel}$$

where:

e_R	Platform position error
e_V	Platform velocity error
e_ψ^{gyro}	Gyro misalignment
$e_{\dot{\psi}}^{gyro}$	Gyro drift
e_{bias}^{accel}	Accelerometer bias
e_{sf}^{accel}	Accelerometer scale factor error

$$G_R = \left(3i_R i_R^T - I\right)\frac{\mu}{r^3}$$

$$M_x(V) = \begin{bmatrix} 0 & -V_z & V_y \\ V_z & 0 & -V_x \\ -V_y & V_x & 0 \end{bmatrix}$$

$$M_{diag}(V) = \begin{bmatrix} V_x & 0 & 0 \\ 0 & V_y & 0 \\ 0 & 0 & V_z \end{bmatrix}$$

S	Specific acceleration on the vehicle (excluding gravity)

Like the error terms, the constraints are most easily expressed in a mathematical form, so a list of inequalities is sufficient for many sensors. Some well understood text descriptions may also be given. For example, the mathematical meaning of a clear line-of-sight is known to any linear covariance applications engineer.

Since the IMU does not take measurements with respect to other platforms, it has fewer constraints and no measurement sensitivity, so a different sensor is used to show a sample constraint list. The measurement involves two platforms, so words such as range and

relative acceleration are assumed to be with respect to the other platform. Three sample constraint specifications are:

Clear line-of-sight
Range < 2500 km
Relative acceleration < 5.0 g

The elements of the measurement sensitivity vector have a one-to-one correspondence with the error state vector. Therefore, the measurement sensitivity depends upon the defined error terms. The measurement sensitivity vector is simply specified in traditional vector notation. The error terms are provided first to put the measurement sensitivity vector into perspective. The specification shown is for a 2-way radio range measurement with clock bias and clock drift.

$$\dot{e}_R = e_V$$
$$e_V = G_R e_R$$
$$\dot{e}_B = e_D$$
$$\dot{e}_D = \left(-\frac{1}{\tau}\right) e_D + u_D$$

$$\mathbf{b} = \begin{bmatrix} \vdots \\ u_{LOS} \\ 0 \\ \vdots \\ -u_{LOS} \\ 0 \\ 1 \\ 0 \\ \vdots \end{bmatrix}$$

where:

$$u_{LOS} = \text{unit}(r_2 - r_1)$$

The measurement sensitivity calculation has non-zero elements only for the sensor taking the measurement, the host platform position and velocity errors of the sensor taking the measurement, and the position and velocity errors of any platform the sensor is communicating with.

The measurement variance is simply an equation that describes the accuracy of the measurement being taken. This may be a constant or a function of the conditions under

which the measurement is performed. For example, a range measurement may decrease in accuracy as the bodies get farther apart, so a sample equation is:

$$\alpha^2_{meas} = 10^{-6} |r_2 - r_1| \sigma^2$$

where σ^2 is a measurement of the accuracy of the instrument. The measurement accuracy variance, α^2_{meas} is the overall measurement accuracy which degrades linearly with increasing distance between the two bodies involved in the measurement. This type of measurement variance is characteristic of a constant power system.

3.2 Mission Specification

The goal of a design specification language for a linear covariance analysis mission is to simply and clearly specify the mission. The language does not need to be restricted to pencil and paper. Rather, it can and should take advantage of advancing knowledge capture and information management technology. For linear covariance analysis problems, using hierarchical objects yields a much more elegant design language.

A hierarchical object language consists of a root object, in which several other objects are defined. Each of these objects then becomes a root object for another set of objects. Each sub-object may capture design information in a different way. For example, one object may simply capture a list of equations while another takes video data. Thus, the hierarchical approach is the most natural method of capturing the inherently hierarchical nature of a linear covariance analysis mission while retaining full flexibility in specific knowledge capture techniques. A diagram of the mission specification object hierarchy is shown in Figure 3-1.

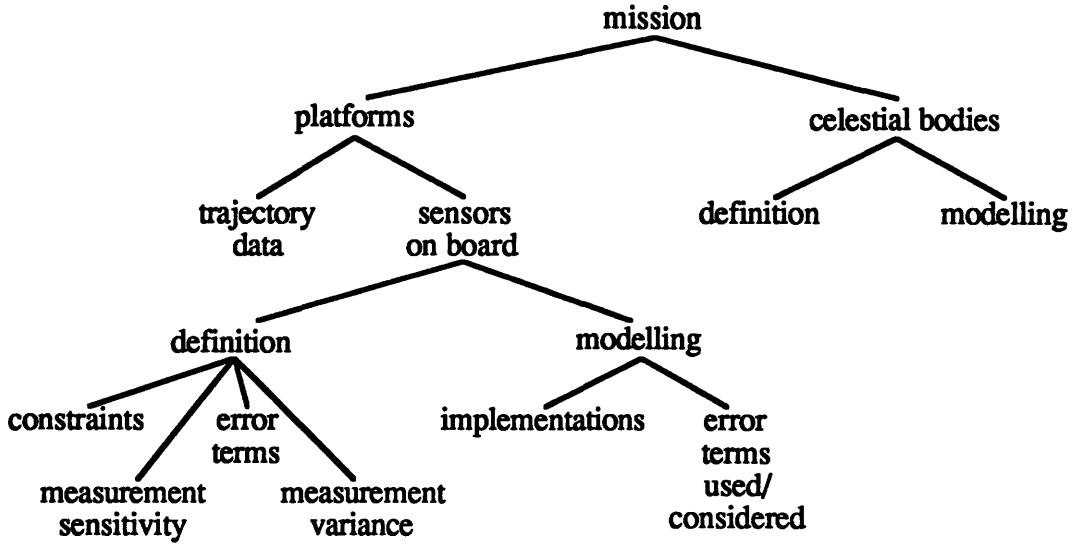


Figure 3-1: Linear covariance analysis mission definition design language object hierarchy.

For linear covariance analysis, the root object is a mission. Each mission is comprised of the platforms and celestial bodies used in the design. The basic mission components are best represented graphically as shown in Figure 3-2. This method clearly identifies the platforms and celestial bodies. For missions with few orbiters and maneuvering vehicles, the trajectories may even be included in the diagrams. The graphical information also provides some hints as to the relationship among the bodies, although the specific data is included as a part of the body objects.

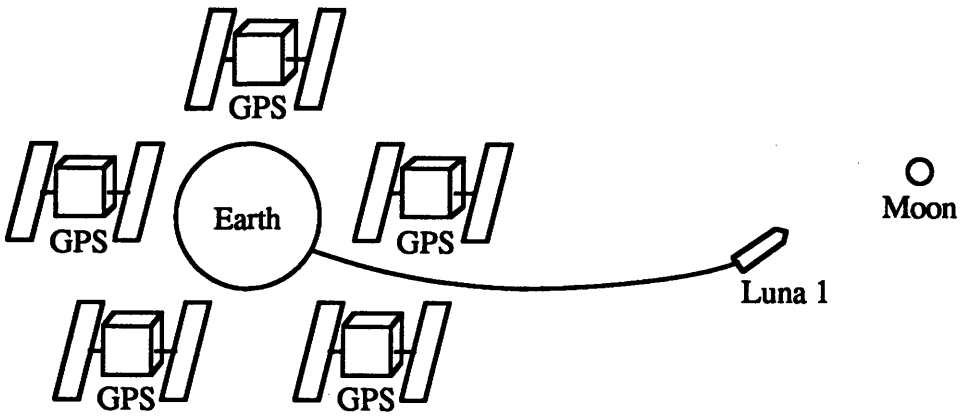


Figure 3-2 Mission specification design language.

Each celestial body is comprised of a celestial body definition and modelling selections. The purpose of the celestial body definition is to provide the necessary information to model a celestial body such as Earth. In some cases, data for several different models may

be provided. For example, a celestial body definition may include many harmonics of a celestial body which are unnecessary in a particular problem. Therefore, the celestial body object used in a mission must consist of a definition as well as modelling selections such as the gravity model. The celestial body definition is described in Section 2.1.1. As Figure 3-2 shows, the celestial body definition is identified by the graphical symbol used to represent the body. The additional modelling information is simply listed. For the scope of the linear covariance analysis specification problem under consideration, the only additional modelling parameter is the gravity model.

The second mission object, platforms, include the type of platform, sensors on board, and trajectory data. Each platform is identified by a name which is displayed along with the platform graphical symbol which shows the type of the platform. The method of specifying the trajectory data depends upon the type of platform. For maneuvering vehicles and rovers, the trajectory is given by a mathematical equation or a series of vectors that contain the state information for each time step. The orbiter trajectory is defined by a series of initial orbital elements and a central body which are specified in the same manner as those of the celestial body definition. Finally, latitude, longitude, and altitude and host planet information for beacons are used to determine their trajectory. Beacons also have a unique modelling parameter, elevation angle. The elevation angle is used to determine when sensors on board the beacon can communicate with other sensors both on the planet or above it. A sample beacon specification is provided:

Name: Beacon-1		
Type: Beacon		
Sensors on board:	Radio	Litton A-190
	2-way range	Honeywell HT-525
Central body:	Earth	
Latitude:	41.75°	
Longitude:	-125.12°	
Altitude:	1.342 km.	
Elevation angle	8°	

The list of sensors on board is comprised of sensor objects which form another level of the hierarchy. Like celestial bodies, the sensor object is made up of a sensor definition and additional modelling information. This includes the sensor implementation, error term modelling, consider states, and measurement definitions. Since sensor definitions are used only to capture the functional form of the differential equations that define the error propagation, implementations are used to gather the specific performance parameters for a

particular piece of hardware. Since the sensor on board the platform is a real piece of hardware, both a definition and an implementation are required. The design also may not require as high a modelling fidelity for a sensor as is provided in the definition, so the sensor object includes information telling which error terms should actually be included in the model. These consider states are error terms which are propagated but not used to improve measurement accuracy. A sample sensor object specification is shown below:

Name (Definition): 2-way range
Implementation: Litton CC-121

Fully modelled error terms:	Clock bias Delay bias
Consider states:	Transceiver clock drift

The final sensor object element is a measurement definition. This definition is itself an object that identifies which measurements are taking place and when they should occur. Each sensor can only take one measurement. Therefore, the only information about a measurement that needs to be entered is the sensor taking the measurement, and the actual bodies that are referred to by the associated body definitions. Measurement timing information includes the measurement frequency and a list of times the sensor is turned off.

Estimate range to:	Space Shuttle Atlantis Beacon-1 TDRSS-1 TDRSS-2 TDRSS-4 GPS-1 GPS-4 GPS-11
Measurement frequency:	5 hz.
Shut off:	4.0 min. to 5.2 min.

Chapter 4: Automated Linear Covariance Analysis Tool

The purpose of developing an automated system for linear covariance analysis is twofold. First, the specification of a high-level design language as compared to low-level computer code is much easier for an engineer. As with any process, this specification simplicity leads to a more reliable process. Second, automation dramatically reduces the time necessary to produce simulation code and even evaluate the results. Thus, the ease of specification, increased reliability, and faster development of simulation code and analysis all contribute to an ability to produce linear covariance analysis simulation code in a more cost-effective way.

Automation of the linear covariance analysis tool begins by specifying the system design through a user interface. This interface is designed to be as close as possible to the natural design language described in Chapter 3. The implemented interface is discussed in Section 4-1. The linear analysis tool then reads the data entered by the engineer and places it into a set of data structures. These structures are defined in detail in Section 4-3. Finally, an understanding of linear covariance analysis and its application to navigation system evaluation is combined with the user specified data to produce the actual simulation code.

The code generation process is accomplished with the help of an engineering software development tool called ASTER (Automation Software Technology for Engineering Reliability). ASTER is capable of taking equations and block diagrams and converting them into computer code. The choice to use ASTER rather than a code generator built specifically for linear covariance analysis is twofold. First, it eliminates the need to write a code generator, thereby allowing more time to be spent developing a powerful linear covariance analysis specification tool. Using the ASTER code generators serves the long-

term interests of the linear covariance analysis tool as well. As improvements are made to the ASTER code generation process, the linear covariance analysis tool automatically acquires these benefits. Furthermore, it reduces the amount of maintenance required by the two tools. The software development process for linear covariance analysis problems is shown in Figure 4-1. The figure also delineates responsibilities of the engineer, linear covariance analysis tool, and ASTER.

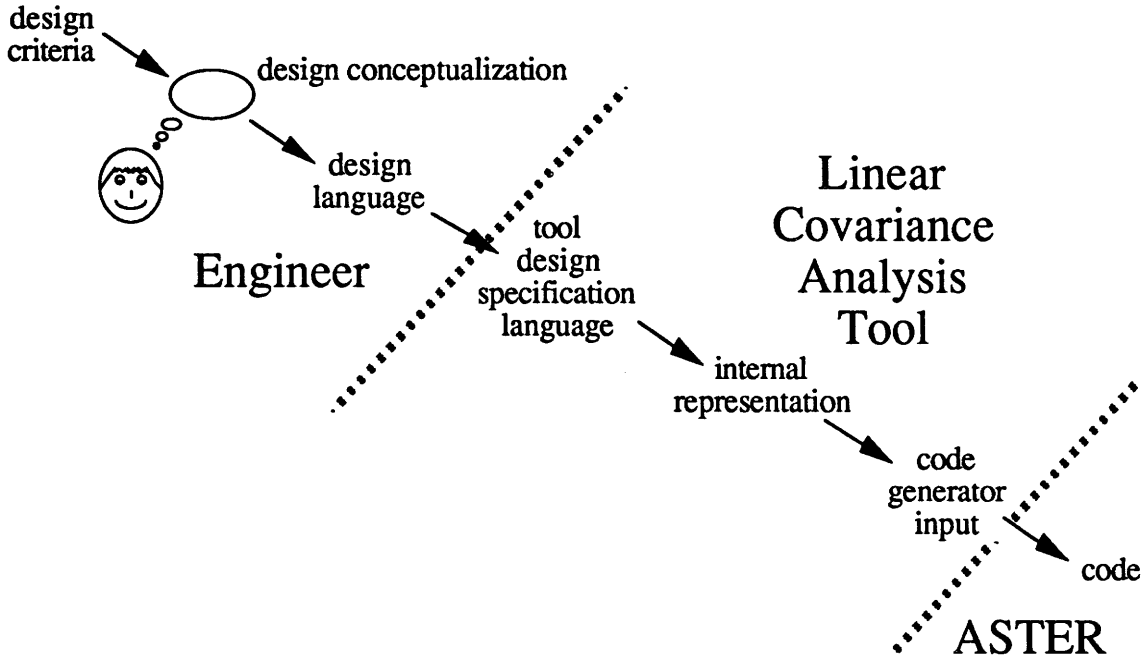


Figure 4-1: Software development process for linear covariance analysis code.

4.1 Mission Specification Language Implementation

The user interface for the linear covariance analysis tool is designed to allow an application engineer to specify linear covariance analysis missions using the language described in Chapter 3. However, some modifications to the design have been necessary in order to simplify development and ease of use. Despite these changes, the implementation of the natural design language in the graphical user interface successfully approximates the language requirements.

Since the specification automation and code generation processes dramatically reduce the amount of time required to develop a linear covariance analysis simulation, the application engineer will now spend the majority of his or her time analyzing results. Therefore, any further reductions in the time required to perform linear covariance analysis must relate to

this analysis. For this reason, the user interface also includes capabilities to aid in the analysis process. In particular, the engineer can specify outputs that are to be examined in a variety of formats. Graphical formats such as plots and charts are valuable for examining time traces, while a textual format is used to manipulate and view the covariance matrix.

The following sections describe the user interface for the automated linear covariance analysis tool. The windows were designed to correspond to the design language as closely as possible. However, some differences are unavoidable due to the processes by which data can be entered into a computer.

4.1.1 Defining Linear Covariance Analysis Elements

Before any mission can be specified, the properties of the elements of the mission must be defined. When applying linear covariance analysis to navigation systems, three main elements are used, celestial bodies, platforms, and sensors.

4.1.1.1 Celestial Bodies

The celestial body dialog box is used to specify the properties of a celestial body. The first data input is the name of the celestial body. Then, the physical properties of the body are specified. These include the gravitational attraction constant μ, radius, atmospheric height, rotation rate, and the model of the body to be used for gravitational calculations. The second section is used to describe the orbital properties of the body. This information is used when more than one celestial body is included in a mission specification. The six orbital elements required by the tool are the semi-major axis, eccentricity, inclination, longitude of the ascending node, argument of pericenter, and true anomaly. In addition, the central body for this orbit must also be given. Finally, the engineer can enter a detailed description of the models used for deriving the numbers specified to the linear covariance analysis tool. The celestial body definition dialog box is shown in Figure 4-2.

4.1.1.2 Platforms

In order to limit the scope of this effort, platforms are the only basic linear covariance analysis element that cannot be specified by the user. Since the user cannot define the properties of a platform type, no user interface is necessary. However, the linear covariance analysis tool provides four types of platforms: maneuvering vehicles, rovers, orbiters, and beacons.

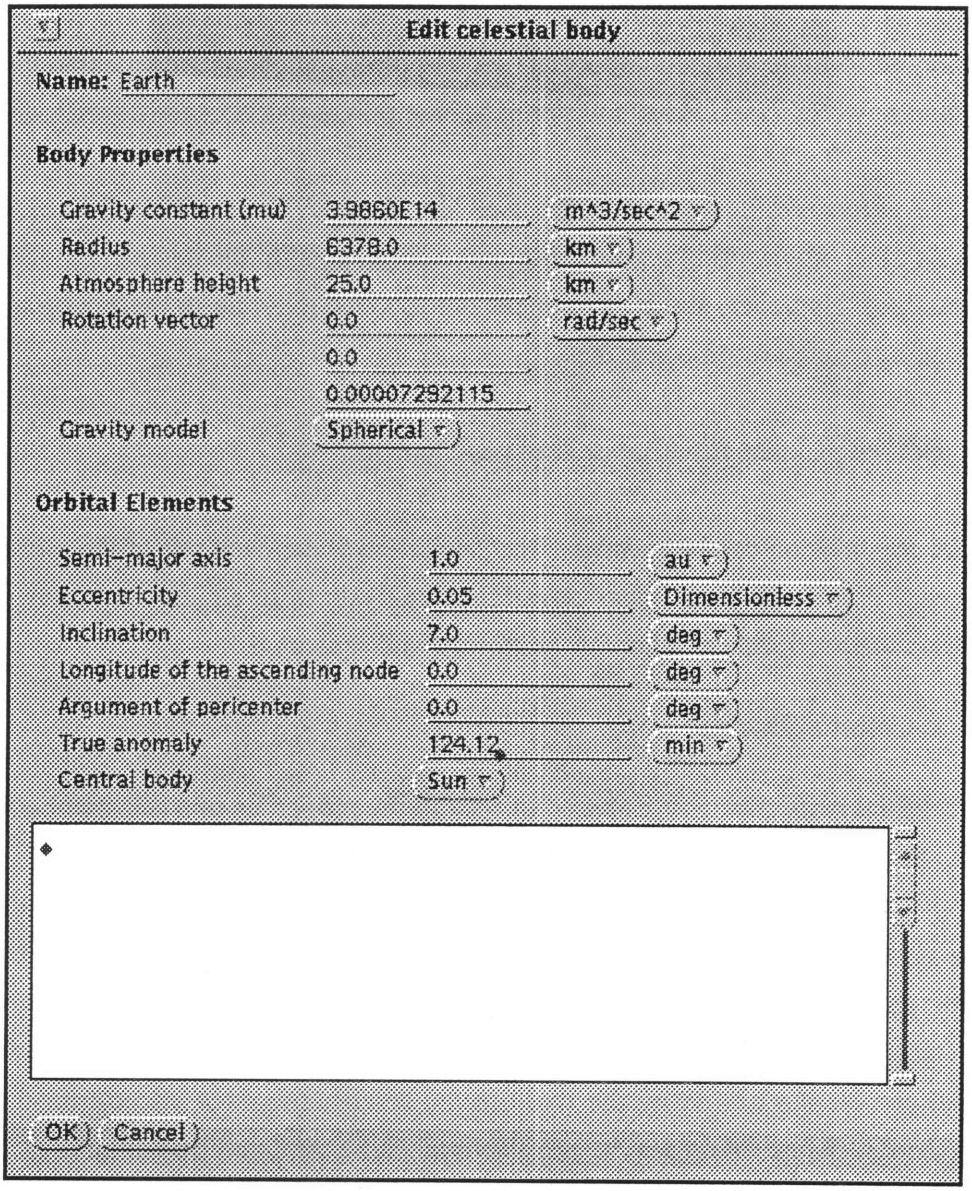


Figure 4-2: Celestial body definition window.

Maneuvering vehicles have self-propulsion capabilities. Therefore, the applications engineer must specify the trajectory of the vehicle. Rovers can also propel themselves, but they must reside upon the surface of a celestial body. A trajectory must also be provided for rovers. Orbiters move in a fixed orbit about a celestial body; they may have an attitude control system, but the Δv introduced by that system is ignored. Once the simulation has begun, the orbit is determined by the gravity of the central planet, but an initial orbit must be specified. Beacons are a navigational aide that reside at a fixed location on a celestial body. A beacon is defined by its location on the body which includes its latitude, longitude, and altitude.

4.1.1.3 Sensors

As expected, sensor models are the most difficult to specify since they are the most complicated. The sensor model is actually a template for a class of sensors. In most cases, the equations that describe the error terms of a sensor have the same functional form regardless of the manufacturer of that sensor. The performance differences between the sensors are captured by entering different performance constants for each vendor's product. In the linear covariance analysis tool, the term sensor refers to the general functional description of an instrument model, and implementation refers to the model for a specific piece of hardware. This model consists of the general description combined with the vendor-specific performance parameters.

Sensors introduce an interesting knowledge capture situation. Much of the modelling of a sensor is described with equations or inequalities. To someone familiar with linear covariance analysis, the equation

$$\dot{e}_{sf}^{gyro} = \left(-\frac{1}{\tau_{sf}^{gyro}}\right) e_{sf}^{gyro} + u_{sf}^{gyro}$$

quite clearly represents the differential equation for the scale factor error of a gyro. Similarly,

$$alt = magnitude(r_{s/c}^{i} - r_{earth}^{i}) - r_{eq}$$

calculates the altitude of a spacecraft above the earth from the inertial position vectors and the equatorial radius of the Earth. However, to the linear covariance analysis tool, these equations are just a set of mathematical operations applied to some variables. The variables have no physical meaning. Therefore, the meaning of each input of an equation must be given. In other words, the user must tell the linear covariance analysis tool that e_{sf}^{gyro} is the gyro scale factor error, $r_{s/c}^{i}$ is the inertial position of the spacecraft, r_{earth}^{i} is the inertial position of the Earth, r_{eq} is the equatorial radius of the Earth, and so on.

When an equation is entered into the linear covariance analysis tool, the equation is parsed and the inputs are displayed in a list box. Each input gets its name from the variable name, but the type, units, frame of reference, and description are supplied by the user. The other

slot, source, is used to identify the meaning of each input. The allowable quantities are constants, mission loads, state vectors of the host body or an associated body, celestial body properties, and simulation parameters.

A constant is a global constant such as π, 6.0, or e. These only need to be defined once per mission. A mission load is constant during a simulation, but the constant can have different values for different implementations of the sensor. State vector information includes the position, velocity, acceleration, attitude, angular velocity, and angular acceleration vectors. The available celestial body properties are the gravitational attraction constant μ, radius, atmosphere height, rotation rate, and the initial orbital parameters semi-major axis, eccentricity, inclination, longitude of the ascending node, argument of pericenter, and true anomaly. Direction cosine matrices describe a vector expressed in one frame of reference in another frame of reference. Finally, the simulation parameters include the current simulation time, start time, end time, time step, and the measurement underweighting. The sensor definition process requires several dialog boxes. The main dialog box is used to enter identification information, create error terms and constraints, and to call other dialog boxes for defining error terms, measurement sensitivity, constraints, measurement variance, and contributions to the host platform velocity error.

The identification information available for a sensor are a name, label, and description. The name is used in all the menus throughout the system to identify the sensor. The label is an abbreviated version of the name used in the sensor palette on the right hand side of the main window. The description is entered by clicking on the button "Sensor description..." at the bottom of the dialog box. This pops up a dialog box that enables basic text editing. This space can be used to describe the modelling process used to derive the inputs, the source of the input data, or any other useful background information. The sensor dialog box is shown in Figure 4-3 for a 2-way radio range sensor.

The first list box is used to create, copy, and delete error terms. When an error term is selected from the list, its name and description are shown to the right of the list. In addition, the engineer specifies the error term equation and the measurement sensitivity equation. These items are selected from a list of options. For the error term definition, two options are available, general equations or a first order Markov model. The first order Markov model is supplied simply as a way of speeding data input. A Markov model can be entered as a general equation. If a different general equation is desired, a new dialog box appears on the screen.

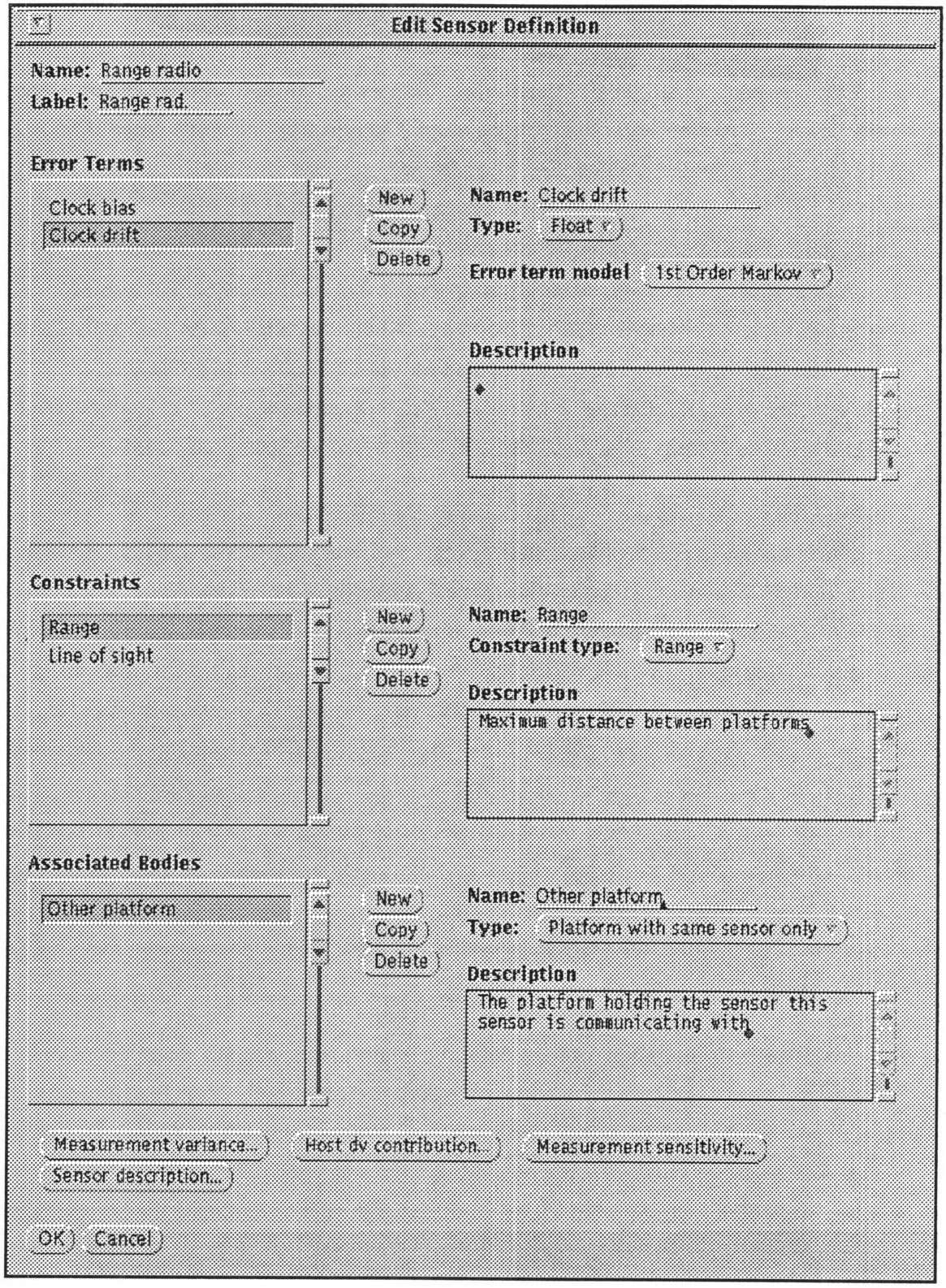


Figure 4-3: Sensor definition window.

The second list box shows the operating constraints placed on the sensor. Constraints can be created, copied, and deleted by using the buttons to the right of the list box. When a constraint is selected from the list box, its name and description can be entered. Also, the constraint itself is chosen. The linear covariance analysis tool provides three constraint options: line-of-sight, range, and a general constraint expression. A line-of-sight constraint is used to check whether the sensor has a clear line to the sensor it is trying to communicate with. A range constraint allows the engineer to specify the minimum and maximum distance between the sensor and either the body or sensor it is taking a measurement with. Finally, the general equation can be used to enter any other constraints. Both the line-of-sight and range constraints can be entered with the general equation capability, but they are provided as special options to reduce specification time.

The final list box shows the associated bodies for this sensor. An associated body is used as a placeholder for other objects during the specification process. When the sensor is being modelled, the exact body the measurement will be taken from is not known. Therefore, an associated body is used to identify the fact that another body is involved without identifying which body it is. For example, an error term for an altitude measurement device on board the space shuttle may require the distance from the shuttle to Earth. This is computed simply by taking the magnitude of the difference between the inertial location of the body and the inertial location of the center of Earth and subtracting the Earth's radius. However, when the sensor is being defined, the linear covariance analysis tool has no knowledge of a planet Earth, so an associated body is defined to identify that a planet's properties will be involved in the sensor model. When a measurement is defined, the associated bodies are replaced by the actual bodies used in the measurement.

An associated body definition consists of three components, a name, description, and type. The tool provides four types of associated bodies: celestial body only, platform with same sensor only, platform only, and any. This is used to determine which bodies are eligible to replace the associated body when the measurement is defined. For example, a radio cannot communicate with a planet, it must communicate with another radio, so the associated body type would be a platform with same sensor.

Finally, the main sensor modelling window has five buttons that enable the user to specify the measurement variance, contribution to the velocity error of the host platform, measurement sensitivity of the host position and velocity error, and a detailed description of the sensor and its model.

Error Term Definition

Although an error term definition is a single differential equation, the linear covariance analysis tool breaks the equation into the individual components which are summed to comprise the right hand side of the equation. This is done for two reasons. First, it ensures the linearity of the equation by automatically preventing the user from specifying non-linear elements. Second, it makes the specification process slightly easier for the user.

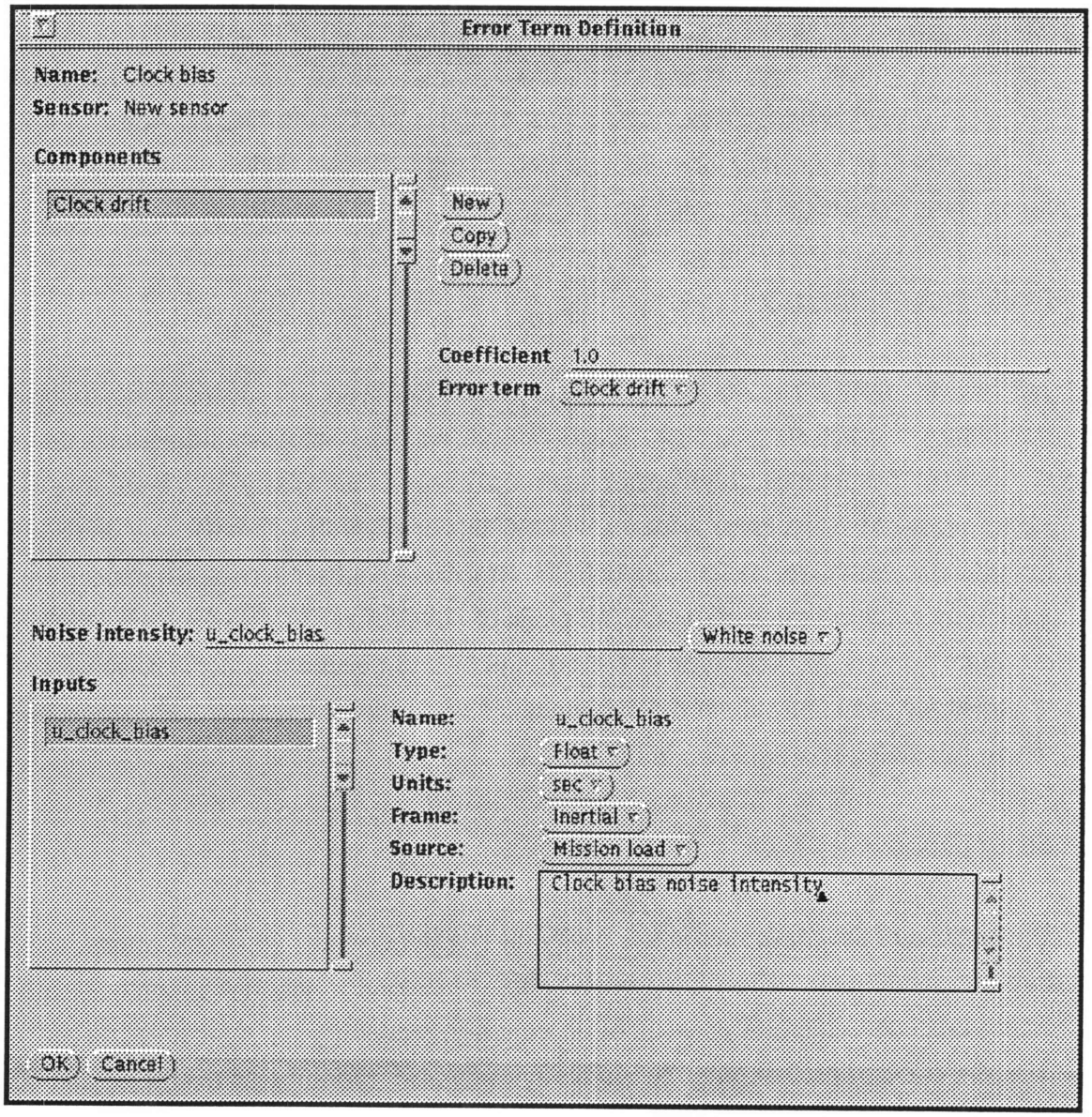


Figure 4-4: Sensor error term definition window.

An error term component consists of a coefficient and the error it is multiplying. The error term can be any error defined for the sensor as well as the position and velocity error of the host platform. The coefficient can include any of the variables outlined at the beginning of this section.

Each error term also has a noise component. Since the linear covariance analysis is a statistical simulation, any noise must be white; however, the intensity of the noise can be specified. The intensity is usually constant, but it can be a function of the physical variables included in the linear covariance analysis. For example, the noise of a range measurement may depend upon the distance between the two bodies. The error term and noise definition dialog box is shown in Figure 4-4.

Measurement Sensitivity

The measurement sensitivity vector is used to relate the errors in a measurement to the error terms in the error state vector. Each error term has a corresponding measurement sensitivity. Sometimes, the measurement sensitivity is not easily expressed as a single equation, so an area is provided to allow the engineer to specify any preliminary calculations which are necessary to define the measurement sensitivity. The measurement sensitivity equation can include the full range of physical variables, which are specified in the same way as the variables in the error term and noise equations. A space is also provided for comments about the measurement sensitivity calculation. The measurement sensitivity dialog box is shown in Figure 4-5.

Constraints

The purpose of the constraints is to identify the conditions under which the sensor is able to take a measurement. This dialog box is almost identical to the measurement sensitivity dialog box. A constraint consists of an equality, inequality, or logical operation. The terms in the comparison may be any of the physical variables or results of additional calculations which can be specified in the supporting equations text field. These equations may also use physical variables. The inputs are identified in the same way as for the other equations. The constraint dialog box is shown in Figure 4-6.

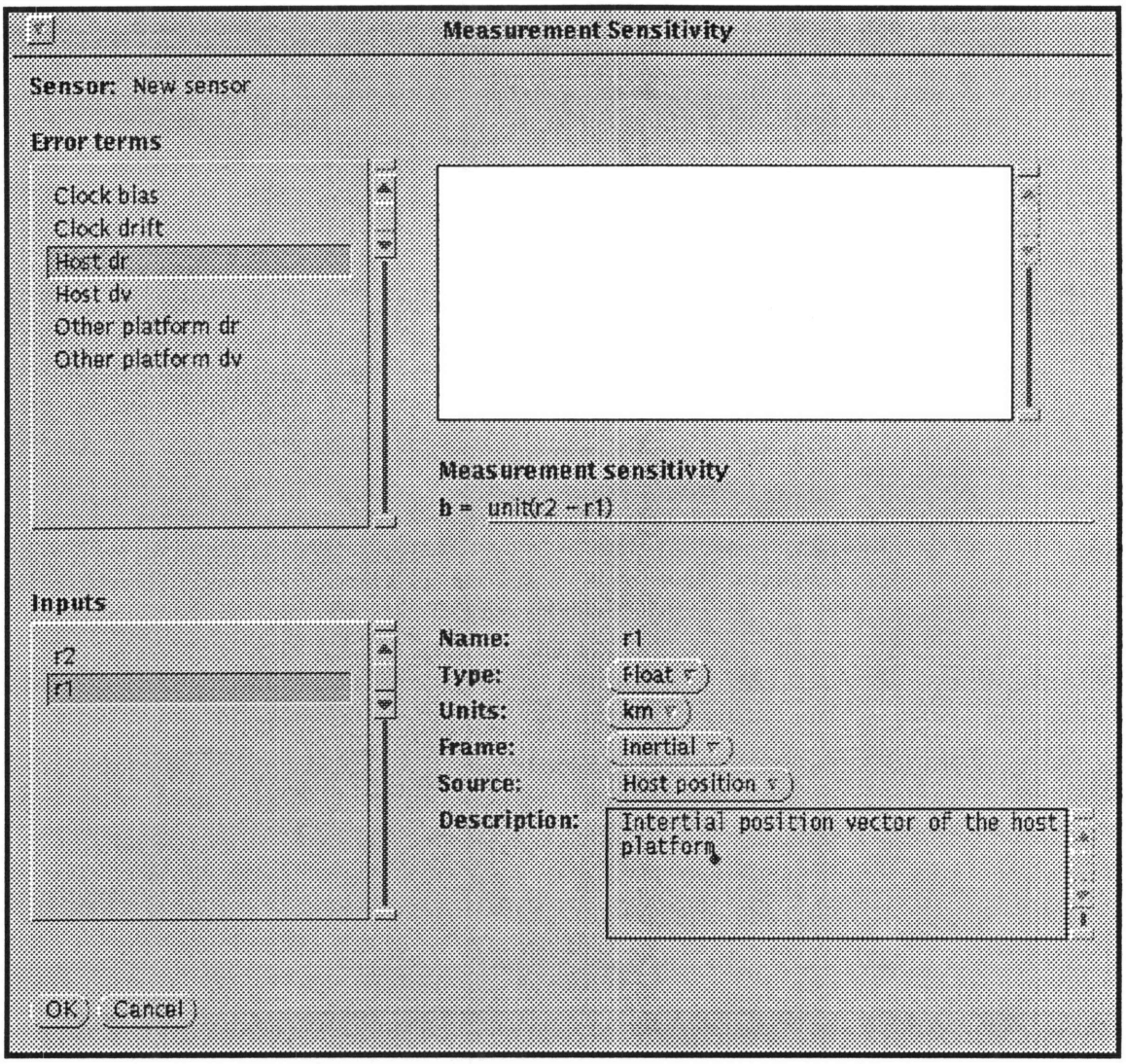


Figure 4-5: Measurement sensitivity specification window.

To speed the specification process, two predefined constraint types are provided. The range constraint sets a maximum distance between the host platform and another body. This body is specified in a small dialog box which appears when the range constraint option is selected. The second constraint type is a line-of-sight constraint. The line-of-sight constraint ensures that the host platform has an unobstructed view of another object. This object is entered in an identical dialog box as for the range constraint. It must be remembered that when a sensor is being defined, access to information about other bodies is accomplished with associated bodies. Therefore, the range and line-of-sight dialog boxes prompt for an associated body, not an actual platform or celestial body.

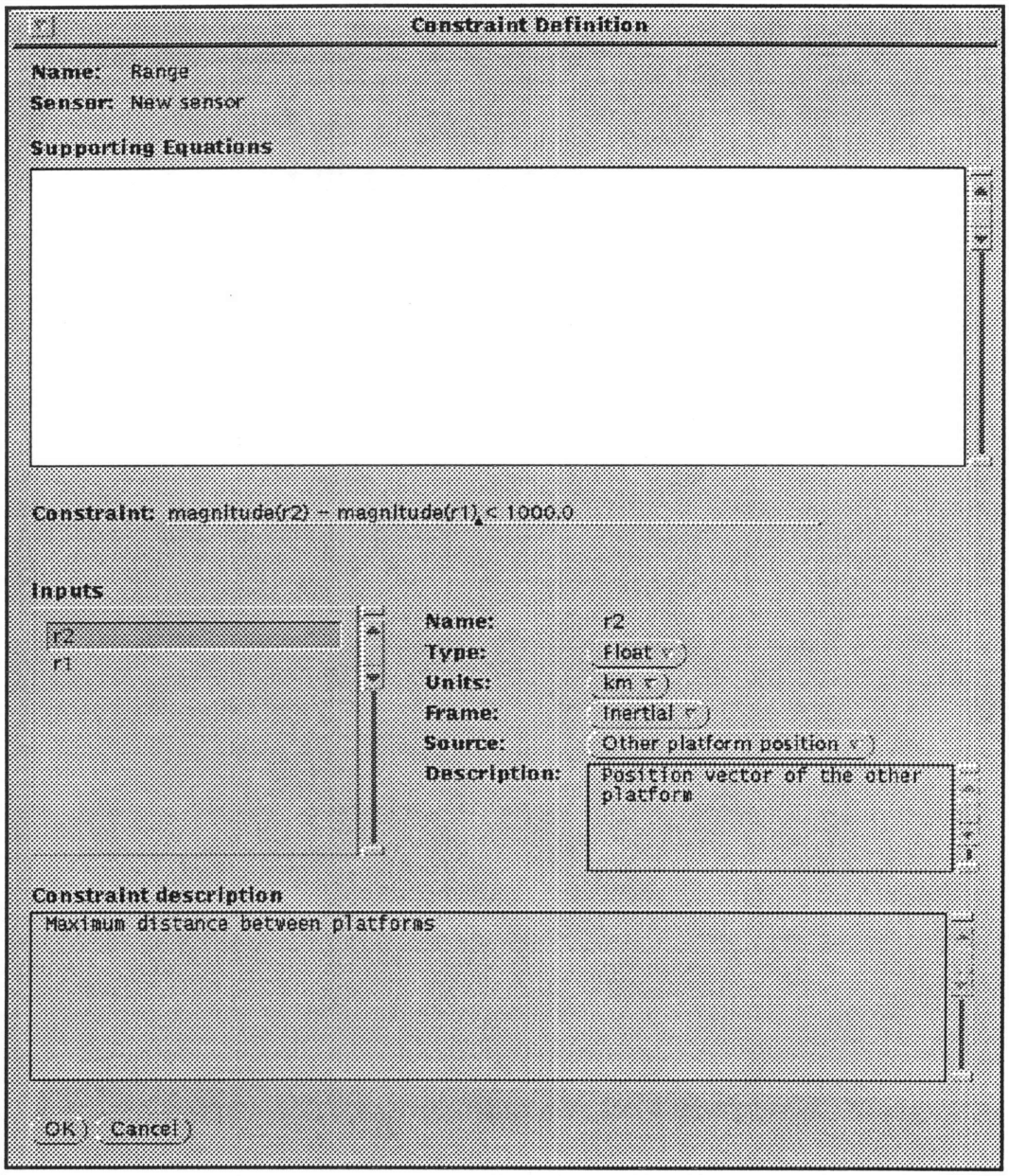


Figure 4-5: Measurement sensitivity specification window.

To speed the specification process, two predefined constraint types are provided. The range constraint sets a maximum distance between the host platform and another body. This body is specified in a small dialog box which appears when the range constraint option is selected. The second constraint type is a line-of-sight constraint. The line-of-sight constraint ensures that the host platform has an unobstructed view of another object. This object is entered in an identical dialog box as for the range constraint. It must be remembered that when a sensor is being defined, access to information about other bodies is accomplished with associated bodies. Therefore, the range and line-of-sight dialog boxes prompt for an associated body, not an actual platform or celestial body.

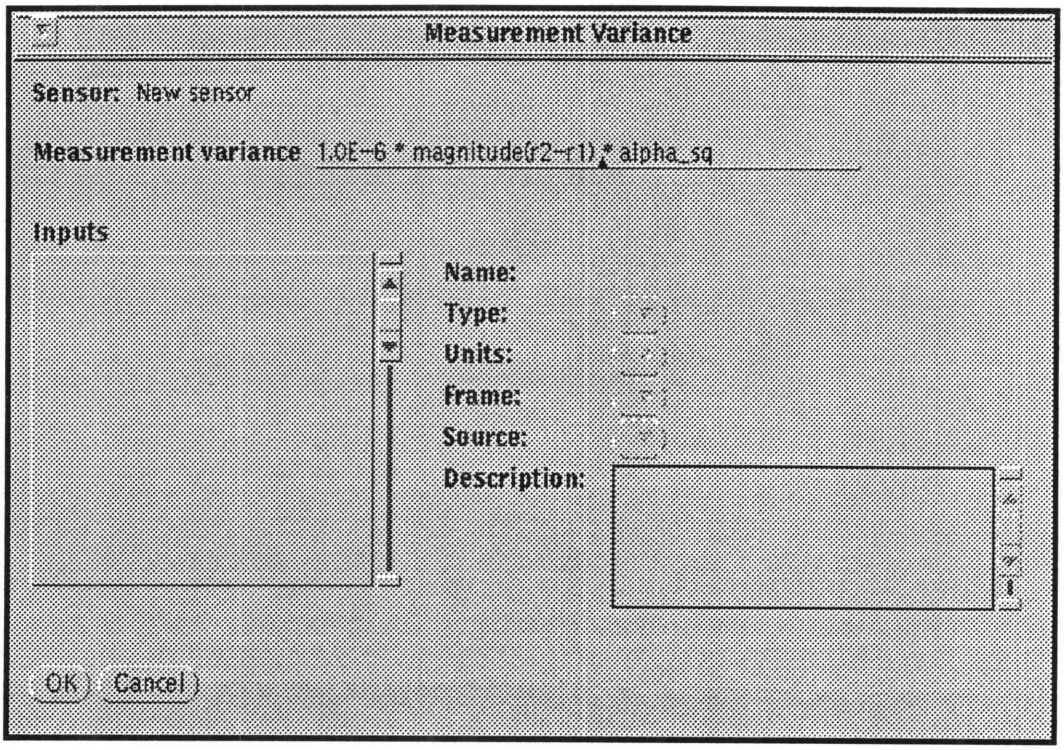


Figure 4-7: Measurement variance specification dialog box.

4.1.2 Specifying the Mission

The first step of the mission specification process is to identify the platforms and celestial bodies involved in the mission. This is accomplished both graphically and with menus. Celestial bodies are added by selecting the appropriate body from the **Add Celestial Body** option in the **Universe** menu. This creates an icon of the body which is placed on the workspace. Platforms are selected from a palette in the upper right-hand corner of the linear covariance analysis window. Once these objects are placed on the screen they can be edited, moved, or deleted with the mouse. The position of the objects on the screen is unimportant to the linear covariance analysis tool, although proper orientation of the objects can help convey physical information about the mission. The main specification window is shown in Figure 4-8.

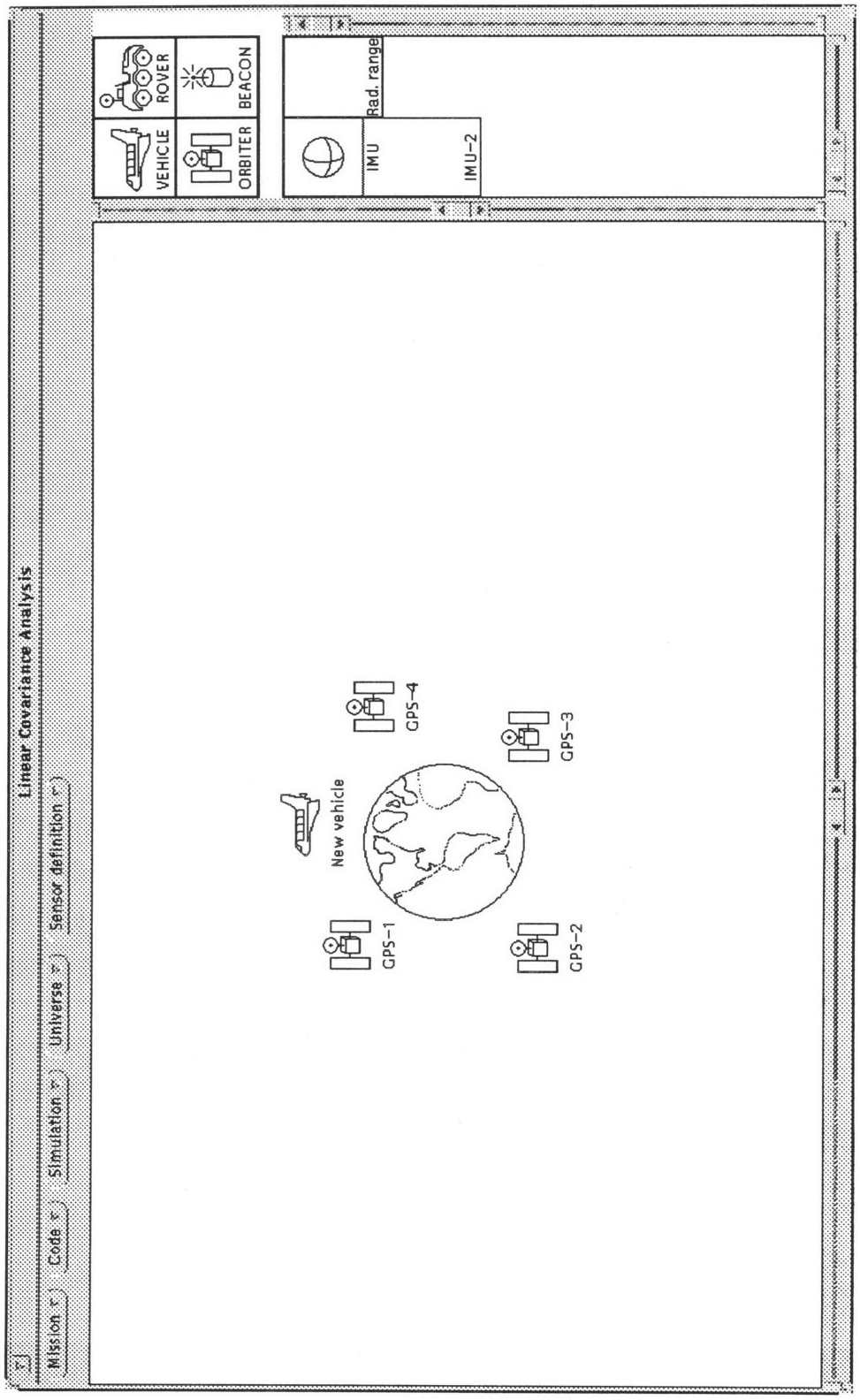


Figure 4-8: Main specification window.

For the celestial body objects, the only parameter not identified is the gravity model. This is done by editing the appropriate celestial body and selecting the desired model from a menu. The platform object specification mechanism varies slightly with the type of platform. All platforms contain a list of sensors on board. A sensor is placed on a platform by selecting the sensor from the sensor palette on the right-hand side of the main window. The sensor is created by left clicking on the sensor in the palette and dragging the sensor onto the desired platform. A sensor is removed from the platform by selecting it from the sensors on board list in the platform object and hitting the **Remove** button.

The method of specifying the trajectory varies among the platform types, so each dialog box is slightly different. Maneuvering vehicles trajectory information is given in the form of a data file. The file name is entered, and the data included is identified by a series of check boxes. The rover dialog box includes a menu of celestial bodies to be used as the central body in addition to the trajectory information. If the trajectory file has the state vectors in inertial coordinates, this slot is unnecessary. Orbiters also have a central body, but the trajectory is specified in the orbiter dialog box. Finally, the beacon dialog box contains a central body menu as well as slots to enter the latitude, longitude, and altitude of the beacon on the central body. A slot is also provided for the elevation angle parameter. Examples of design specifications for each of the platform types are shown in Figures 4-9 to 4-12.

The next level of the hierarchical design language is sensor specifications. These are shown by selecting the **Modelling** button next to the sensors on board list in the platform dialog box. The sensor model consists of three primary pieces of data, the sensor definition, implementation of that definition, and error term modelling. The sensor definition is determined by the sensor that is selected from the sensor palette. When a sensor is selected, it uses the default implementation. This default can be changed by right clicking on the sensor in the palette. This pops up a menu of the available implementations. The implementation can also be selected in the sensor modelling dialog box. Error term modelling consists of which error terms are included in the sensor model and which error terms are used as consider states. The dialog box includes a list of all the error terms in the sensor definition. Each error term is followed by two check boxes, one to include the error term in the sensor model and one to use it only as a consider state. Consider states must have both boxes checked. The error term modelling dialog box is shown in Figure 4-13.

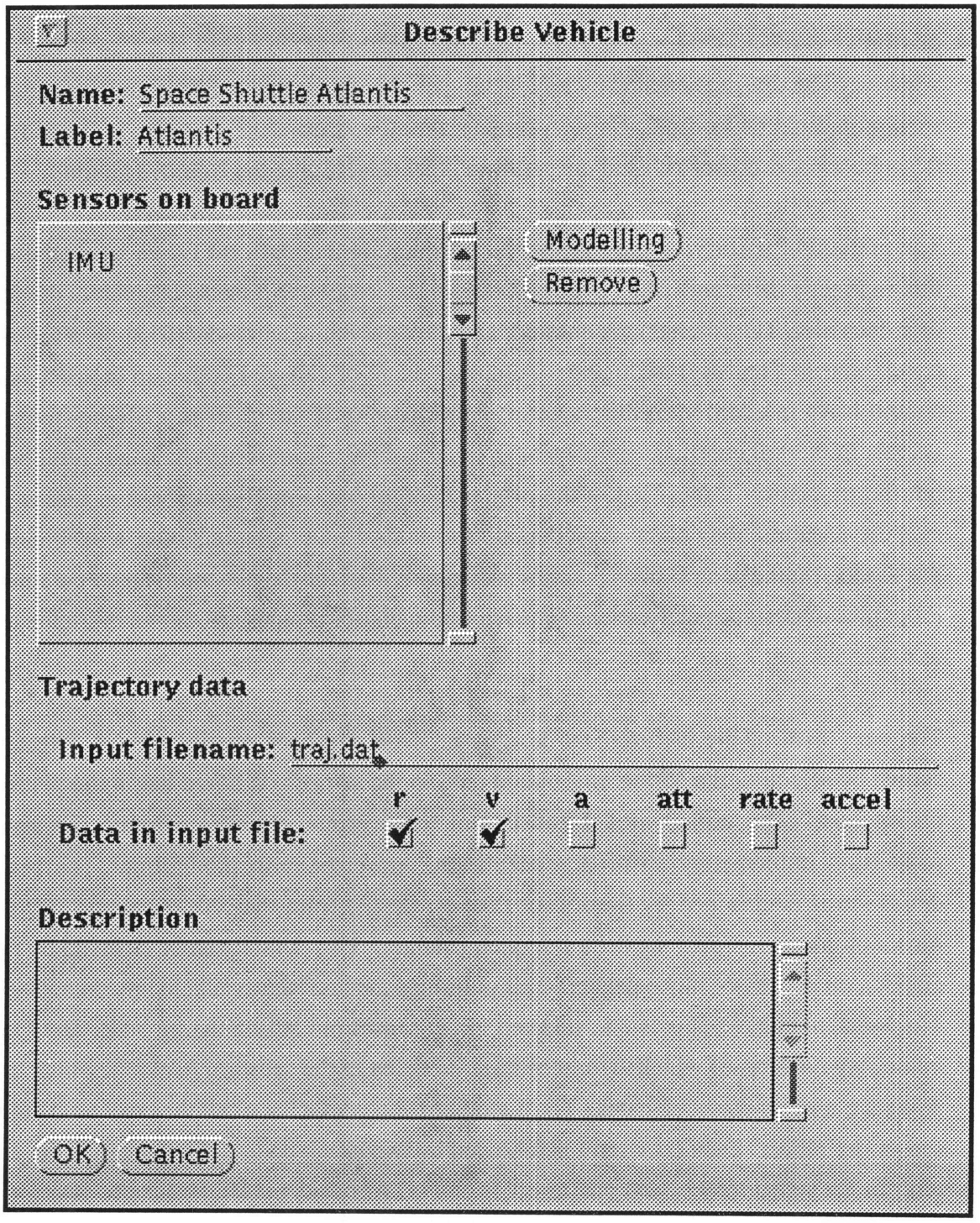


Figure 4-9: Maneuvering vehicle specification window.

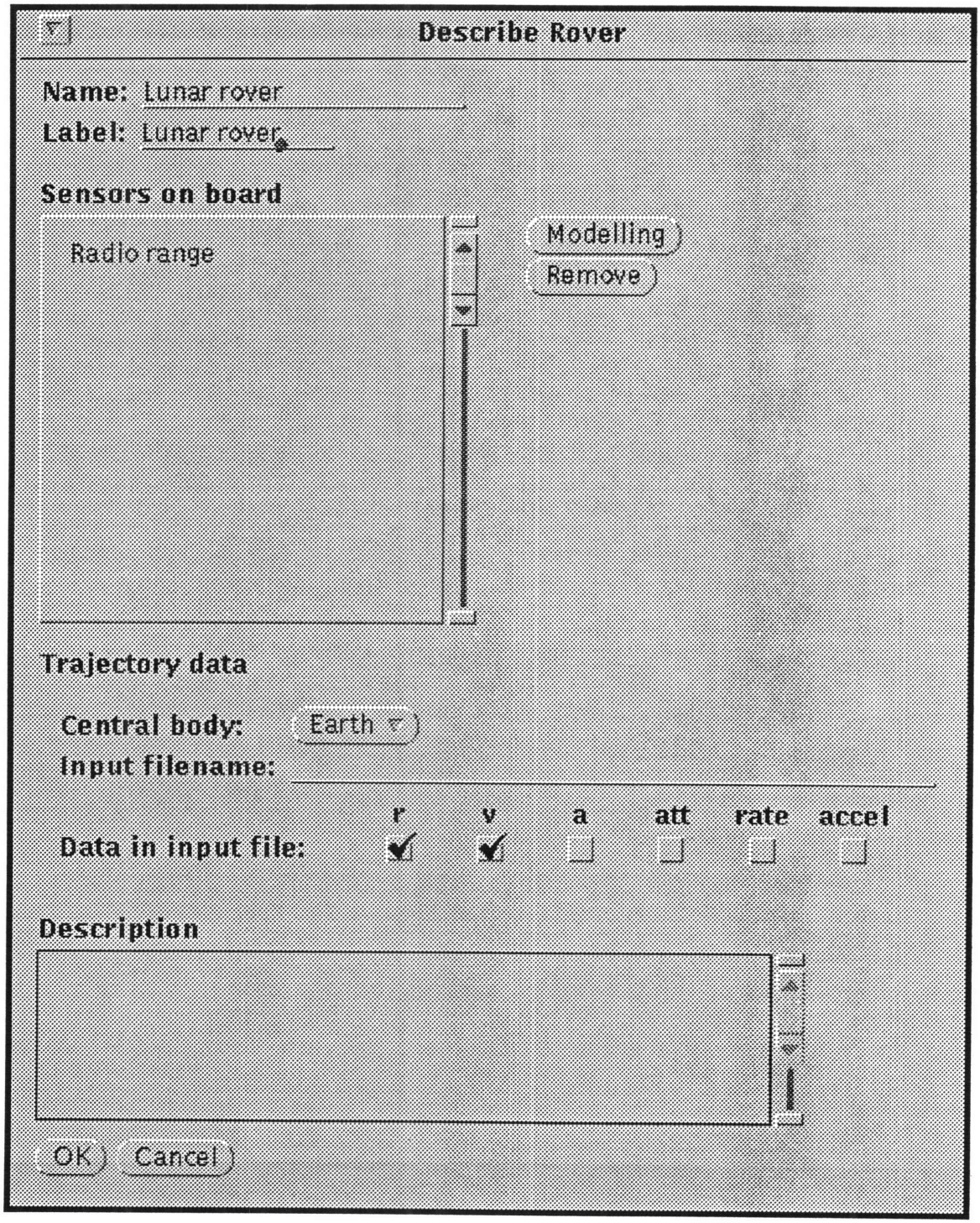


Figure 4-10: Rover specification window.

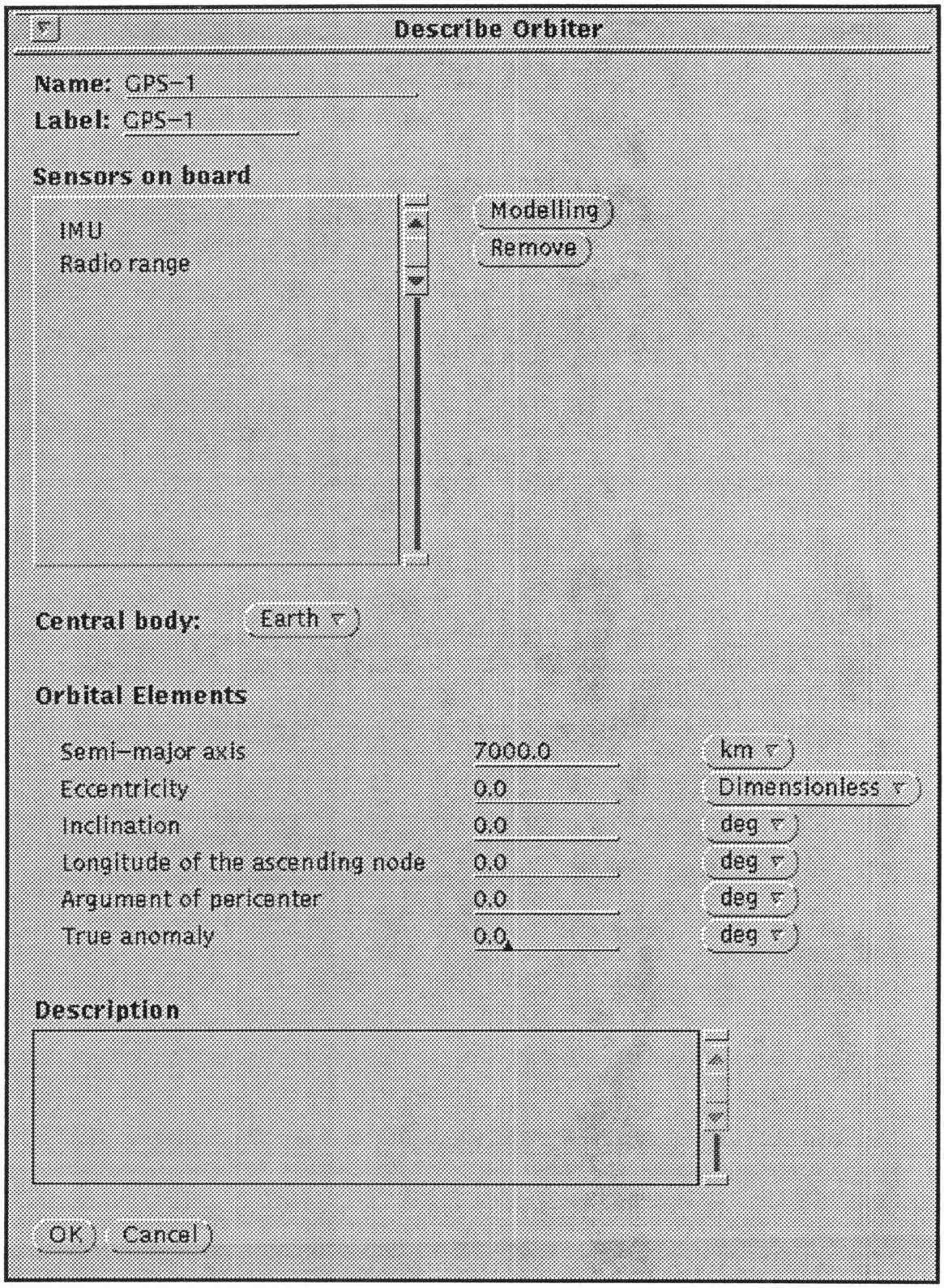


Figure 4-11: Orbiter specification window.

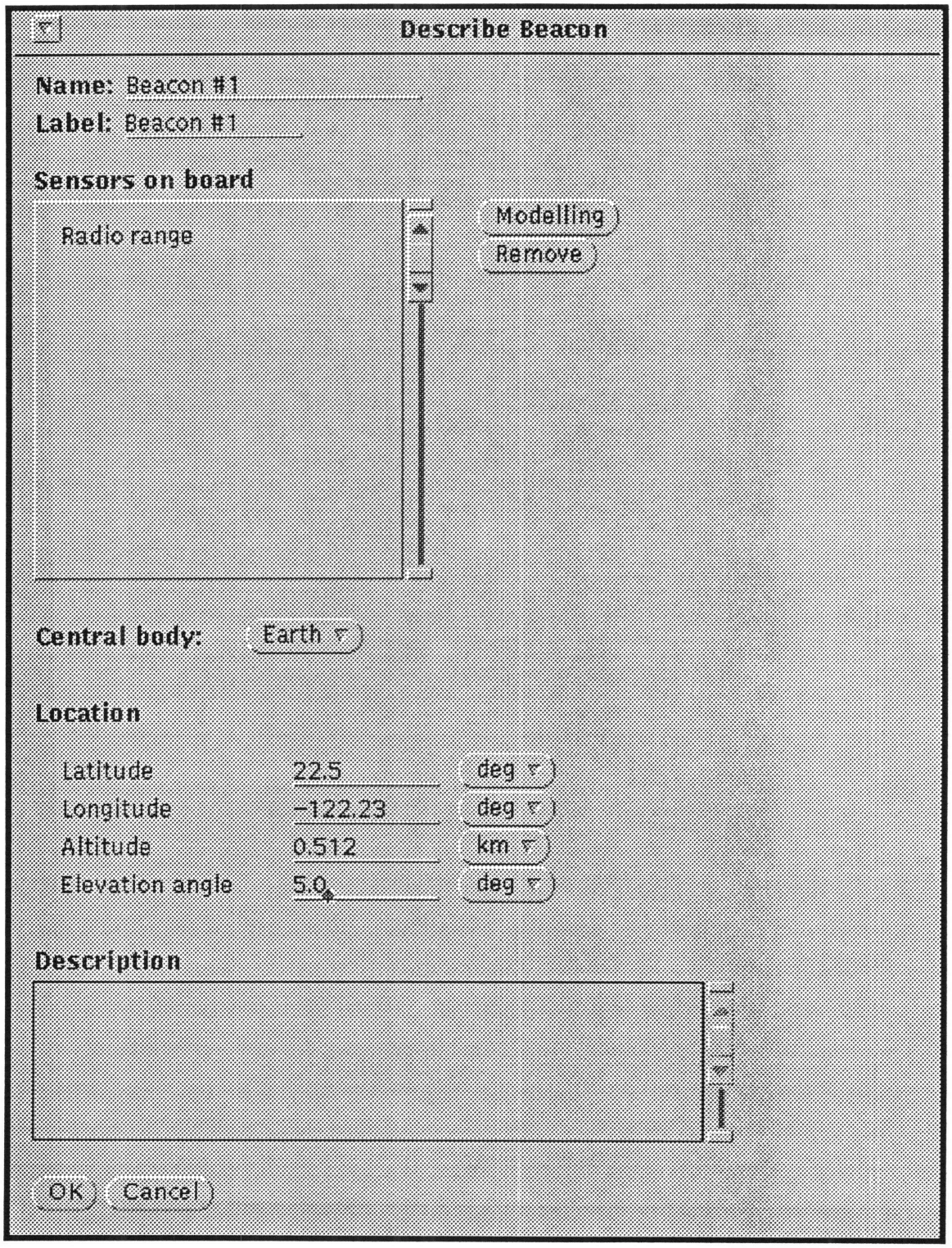


Figure 4-12: Beacon specification window.

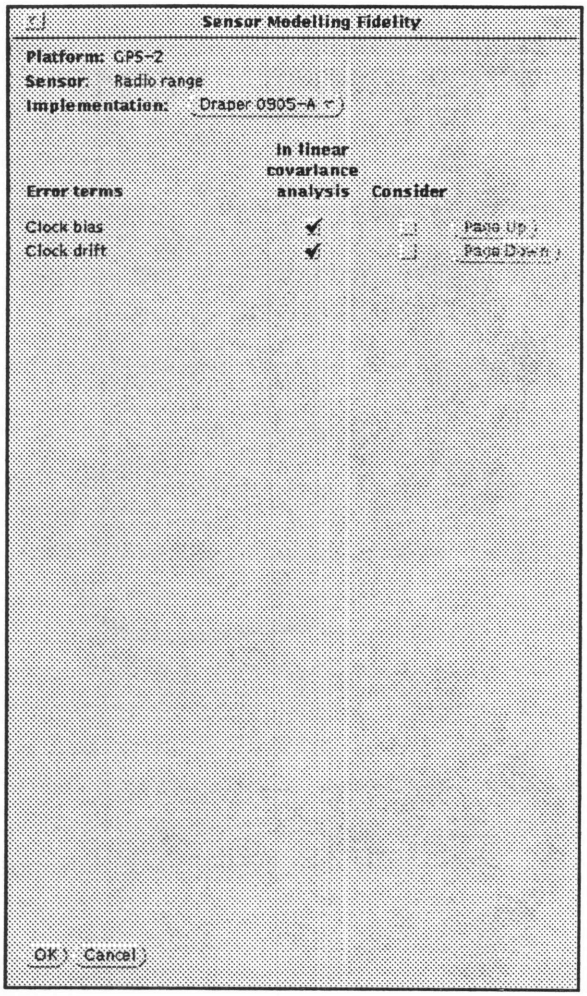


Figure 5-13: Sensor modelling fidelity specification dialog box.

The sensor object also includes a definition of the measurements to be taken and the times at which they should occur. However, the actual implementation breaks from the design specification slightly. Rather than feeding directly from the sensor on the vehicle, the measurements are defined in a dialog box called by the **Measurement definition** option in the **Simulation** menu. The measurements to be taken are defined by identifying which actual bodies replace the associated bodies in the sensor definition. The actual bodies which can be selected are governed by the type of the associated body. The list is defined by selecting from among the eligible objects which are displayed in a menu button. This must be done for each associated body definition. The active associated body definition is selected from a menu item in the dialog box. The measurement frequency is also entered in this dialog box. Finally, spaces are provided for up to four periods where the sensor is turned off.

4.1.3 Analysis Aides

Although an analysis definition is not formally developed here for a linear covariance analysis problem, a user interface has been designed to define the types of outputs that will be examined and the format in which they are viewed. The two most common output modes are two-dimensional plots and textual representations of subsections of the covariance matrix in a variety of coordinate frames.

The 2-D plot is a very general capability for defining graphs of data. The plot is defined by providing equations for the data to be presented along each of the axis of the plot. These equations may be simple parameters such as time, or complicated equations of the linear covariance analysis variables. Any of the physical variables available for sensor and frame specifications can be used in the plotting equations. In addition, the plot can include any of the analysis variables such as the dynamics matrix, covariance matrix, noise matrix, measurement residual variance, and others. The equation may also plot vector-valued quantities. If one axis is a vector and the other is a scalar, separate graphs of each element of the vector versus the scalar are shown. If both axis are vector valued, the elements of each vector are plotted against the corresponding element of the other vector.

The second common output mode is used to examine the covariance matrix. This feature has two forms. First, the engineer can browse the entire covariance matrix at a particular point in time. However, the entire covariance matrix cannot be stored for each pass of a simulation because it would take too much time and memory. Second, a particular sub-matrix of the covariance matrix can be viewed. The sub-matrix is identifyed by selecting two error terms in the error state vector. It does not matter which one is the row and which is the column because the covariance matrix is symmetric. However, the engineer can specify the coordinate frame for each error term. This is important because the meaning of the covariance matrix is usually much easier to determine in a particular frame.

4.2 ASTER

The Automated Software Technology for Engineering Reliability (ASTER) tool is an ongoing effort at the Charles Stark Draper Laboratory to develop an automated environment for engineering application software development, particularly for real-time systems. This environment is designed to use automation to improve the design process and improve reliability while dramatically reducing development time.[2]

One of the key elements of the ASTER design is the array of application user interfaces provided by the ASTER system. These interfaces allow engineers to specify designs with a language specific to the problem they are solving. For example, the current system includes three different input mechanisms: block diagrams, algebraic expressions, and computer code. Each of these input formats is converted into a common representation from which code is generated for either Ada or C. The overall ASTER automated software development concept is depicted in Figure 4-14.

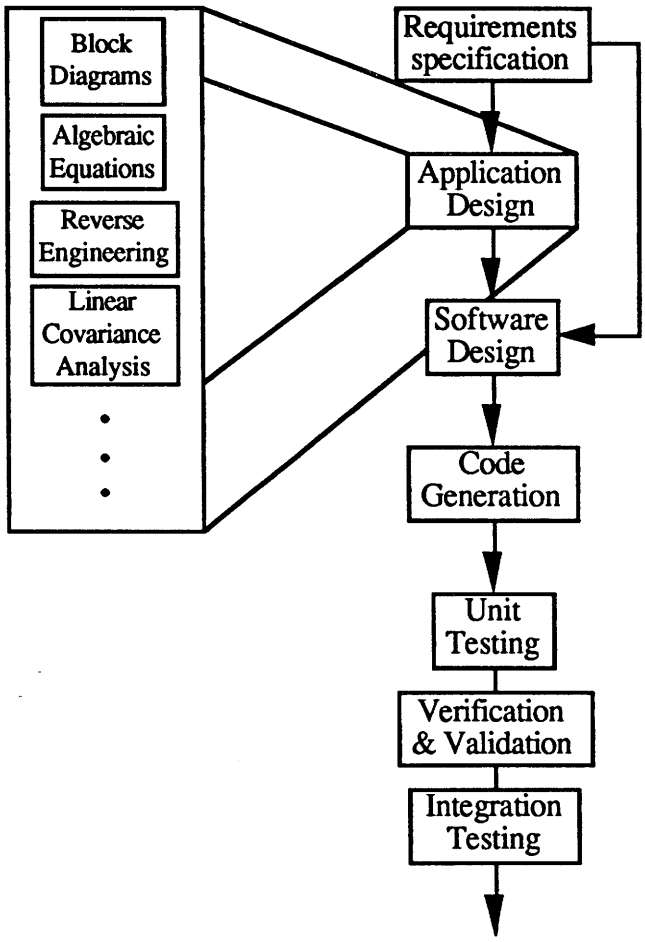


Figure 4-14: ASTER automated software development concept.

Since the linear covariance analysis tool uses ASTER to perform the actual code generation, the input structure to the ASTER code generators plays an important role in the design of the linear covariance analysis tool.

ASTER code generation is based upon control block diagrams. Although control block

diagrams are not the best way of describing the linear covariance analysis algorithm, using the ASTER code generation capability offers two advantages over a system customized to the linear covariance analysis simulation. First, the ASTER code generators already exist and have acquired a level of maturity. Second, and more importantly, it is in the long term interests of the automated linear covariance analysis tool to be part of a common code generation system. If each application specification system had its own code generators, it would be impossible to maintain and upgrade each one. By creating a common interface, namely, the ASTER internal block diagram representation, the same code generators can be used for the full range of application specification tools.

The ASTER block diagram interface is based upon block diagrams from classical control theory.[3] The language has been extended to include elements for a larger body of problems.[4] The block diagrams are organized into functional elements called transforms. A transform takes a series of inputs and "transforms" them into a series of outputs. A sample transform is shown in Figure 4-15.

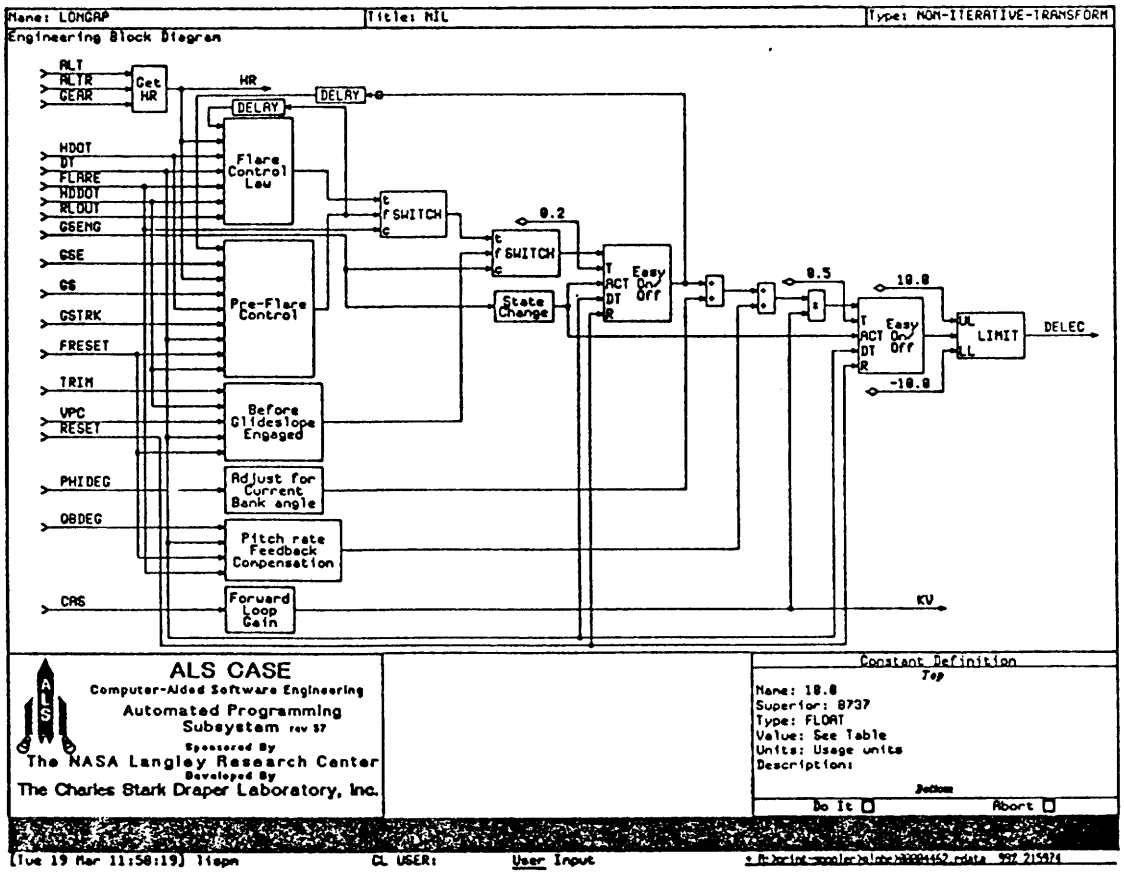


Figure 4-15: Sample transform for the elevator command control logic for a Boeing 737 autopilot landing system.

This type of specification method is inconvenient for implementing equations. For example, the calculation of the distance between two points (x_1,y_1) and (x_2,y_2) is given by

$$z = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

However, in a control block diagram format, this function is much more complicated as shown in Figure 4-16.

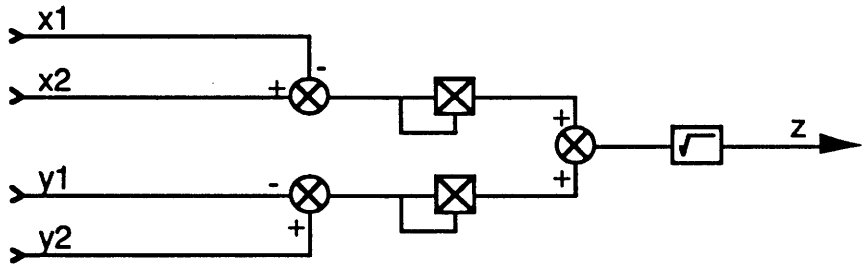


Figure 4-16: Distance equation in a block diagram format.

Despite these difficulties, ASTER provides several advantages. When a specification is mapped into ASTER's block diagram form, the specification can be checked by ASTER for consistency and completeness. Consistency checking includes type checking and ensuring proper connectivity among the items in the block diagram specification. These features aid the debugging process of the ASTER object generation system.

Overall, although the use of the ASTER code generators and the block diagram input specification adds some difficulty to the development of the code generation process for linear covariance analysis simulations, the availability, reliability, and long-term benefits offered by the ASTER system make it worth the extra investment.

4.3 Internal Data Representation

Once the design specification language has been developed, a graphical user interface is designed to enable an application engineer to describe a mission using this specification language. The purpose of the user interface is to provide a mapping from the specification language to an internal data representation. The internal representation of a design is highly dependent upon the programming language used to implement the linear covariance analysis formulation tool. In particular, the representation is constrained by the structures

provided by the implementation language. The closer the structures can match the user interface structure, the easier the transformation process between the user interface and the internal representation will be.

For the linear covariance analysis development tool, object-oriented programming languages such as Common Lisp offer distinct advantages from both a programming and a functional viewpoint. Object-oriented programming enables the functionality of the automated linear covariance analysis tool to remain clear in the internal data representation. Each distinct element of the linear covariance analysis becomes a separate object. Inheritance allows the objects to be classified such that common characteristics such as identification information can be shared among all objects while each objects can still maintain special features unique to that object.[5] This classification system can include many layers. For example, an orbiter might consist of information unique to orbiters in addition to properties common to all platforms and information used by all database objects.

In addition to the functional benefits, object-oriented programming greatly speeds development time. By utilizing inheritance, one piece of code can be written to apply to all objects that draw upon common information. For example, if position and velocity are two properties of a platform, and vehicles, rovers, orbiters, and beacons inherit from platforms, only one function must be written to handle the position and velocity for all types of platforms.

In Common Lisp, data is stored in structures called objects. The properties of an object are defined in a class. A class is comprised of named slots which hold data. A slot is equivalent to a Lisp variable. It can hold anything including lists, strings, symbols, and other objects. When a class is defined, it can inherit some of its slots from other classes. So, if class A is defined to have slots A1 and A2, and class B inherits from A and has its own slot B1, class B will actually consist of slots A1, A2, and B1. In this case A is a superclass or superior to B, and B is a subclass or inferior to A. In a similar manner, functions, called methods, that are written for class A are also then inherited by class B.

The structure of the Lisp objects is designed to mirror the hierarchical specification structure of the natural design language discussed in Chapter 3. Conceptual elements such as missions, sensors, celestial bodies, and platforms each have their own class. Therefore, the mapping from the user interface displays to the Lisp objects is relatively

straightforward.

One minor complication does arise with the linear covariance analysis elements that may not be evident in their description. Several of these elements have two classes associated with them, definitions and manifestations. The difference between definitions and manifestations is best described through an example. Each sensor has a single definition. This definition describes the properties of the sensor including its error terms, measurement sensitivity, and constraints. However, each time the sensor is placed on a platform, it is modelled in a different way. Therefore, a sensor manifestation is used. The manifestation object contains all the specific information relating to the use of that sensor on a specific platform. This includes data such as which implementation is used, which error terms are included, and which error terms are used as consider states. Each sensor definition may have many manifestations.

4.3.1 Common Classes

The internal structure for the linear covariance analysis tool begins with objects available throughout the project. This includes the simplest object containing name and description information, graphical information, variables (signal), coordinate frames, and source data. Source data is used to identify the physical meaning of a signal.

The basis of most objects in the linear covariance analysis is identification information such as a name and description. This class forms the root for most of the linear covariance inheritance hierarchy.

Object: id
Inherits from: saveable-object
Slots: name string
description string

The basic inheritance structures must be augmented with additional classes which are inherited by different classes throughout the inheritance hierarchy. These classes include signal, source and the classes which inherit from it, and orbit. All classes must inherit saveable-object, which is used to enable the object to be written to a file.

Object: signal

Inherits from: id

Slots:	aster-def	ASTER object	Equation used to generate the algebraic transform
	type	ASTER type	Data type of the signal
	frame	<frame>	Coordinate frame (only applicable to vectors)
	units	string	Units of the signal
	source	<source>	Physical meaning of the signal

The signal object is the equivalent of a variable in computer code. The aster-def slot is used during ASTER object generation to point to the ASTER definition of the signal. The source slot tells the linear covariance analysis tool what information the signal actually contains.

Object: source
Inherits from: saveable-object
Slots: None

The source object is used simply as a superclass to help organize the source objects. The objects which inherit from source are constant, mission-load, state-information, host-state, associated-body-state, celestial-body-property, simulation-parameter, and analysis-variable.

Object: constant
Inherits from: source

Slots:	constant-value	string, list of string, or list of list of string	Value of the constant

A constant is usually a perpetual constant such as π or e which does not change its value between equations and is constant in time. The type of the constant-value slot depends upon the type of the input that is constant.

Object: mission-load
Inherits from: source
Slots: None

A mission load is constant for a mission, but its value can be specified at the beginning of the mission. Since almost all signal objects are created during the sensor definition phase, the actual value is not known and is not needed. It is supplied by the implementation. This feature enables the engineer to enter the functional form of an equation and specify a series of actual values that correspond to the different versions of that sensor.

Object: state-information

Inherits from: source

Slots:	platform	platform object	The platform whose state vector is desired
	state-vector	symbol	Identifies which state vector is desired

This object is used for getting state information about any platform. The platform slot must be an actual platform object, not an associated body definition. Therefore, this is not normally used until after a mission has been specified.

Object: host-state

Inherits from: source

Slots:	state-vector	symbol	Identifies which state vector is desired

The host state object is used to identify the a state vector of the host platform during sensor definition. A platform slot is unnecessary because the error term the information is used to define knows which sensor it is modelling, and the sensor knows which platform it is placed on.

Object: associated-body-state

Inherits from: source

Slots:	associated-body-definition	symbol	The associated body whose state vector is desired
	state-vector	symbol	Identifies which state vector is desired

Like host-state, the associated-body-state object is used to refer to state information of an associated body. The actual body that is used is determined at code generation time.

Object: celestial-body-property

Inherits from: source

Slots:	celestial-body	<associated-body-definition>	The celestial body whose physical property is desired
	property	symbol	Identifies which property is desired

This object is also used during sensor definition, so the actual celestial body property desired is not known. Therefore, the celestial-body slot is filled with an associated-body-definition object.

Object: simulation-parameter

Inherits from: source

Slots: parameter symbol Desired simulation parameter

Simulation parameters include time, time step, start time, end time, and measurement underweighting data. The desired property is indicated by the appropriate symbol in the parameter slot.

Object: analysis-variable

Inherits from: source

Slots: variable symbol Desired variable from the linear covariance analysis simulation

Analysis variables consist of all the data calculated during the linear covariance analysis simulation that are not explicitly defined in the user interface. These include the dynamics matrix, covariance matrix, noise matrix, and similar quantities.

Object: direction-cosine

Inherits from: source

Slots:	from-frame	<frame>	Initial frame
	to-frame	<frame>	Transformed frame
	IC	list of list of string	Initial value of the matrix

Direction cosine matrices are used to convert a vector from one coordinate frame to another. The direction cosine matrix can be time-dependent.

Object: algebraic-transform

Inherits from: saveable-object

Slots:	equation	string	Equation used to generate the algebraic transform
	algebraic-transform	ASTER algebraic-transform object	Algebraic transform block diagram
	inputs	list of <signal>	Inputs to the algebraic transform

Algebraic transforms are used extensively throughout the system. An algebraic transform is used to convert an equation into the appropriate engineering block diagram.

Object: frame

Inherits from: id

Slots: primary-dir symbol Coordinate axis being defined by the

		primary equation
primary-eq	<algebraic-transform>	Equation for the primary direction vector
secondary-dir	symbol	Coordinate axis being defined by the secondary equation
secondary-eq	<algebraic-transform>	Equation for the secondary direction vector

A frame is defined by two vectors. The primary vector is used to determine the first axis. This axis is given in the primary-dir slot. The second coordinate axis, defined by the secondary-dir slot, is defined by the part of the secondary equation normal to the primary coordinate axis. The third axis is the cross-product of the first two vectors. The coordinate frame is always right-handed.

Object: graphical-object

Inherits from: saveable-object

Slots:			
	graphical-rep	<icon>	Unhighlighted icon
	highlighted-graphical-rep	<icon>	Highlighted icon
	label	string	Label placed below the object
	x	integer	x position of the object in the workspace
	y	integer	y position of the object in the workspace
	size-x	integer	width of the object in pixels
	size-y	integer	height of the object in pixels

The graphical-object object is used to define graphical representation of celestial bodies, platforms, and sensors.

4.3.2 Linear Covariance Analysis Elements

Celestial bodies

A celestial body object consists of information to identify the celestial body, define the orbit, capture the physical properties of the body, and describe the graphical properties of the body.

Object: celestial-body

Inherits from: id, orbit, graphical-object

Slots:			
	gravity-mu	string	Gravitational attraction of the body
	radius	string	Radius of the body
	atmos-height	string	Atmospheric height of the body
	rotation-rate	string	Rotation rate of the body
	gravity-model	symbol	Model of the body used for gravity computations
	frame-definition	<frame>	Definition of the body frame

Platforms

Platforms start with a general description of data common to all platform types. Then, each type has its own object which inherits from the platform object which contains the information specific to that platform type.

Object: platform

Inherits from: id, graphical-object

Slots:	sensors-on-board	list of <sensor-manifestation>	Sensors on board the platform
	delta-r-dot	<error-term-definition>	Position error differential equation
	delta-v-dot	<error-term-definition>	Velocity error differential equation

Object: vehicle

Inherits from: platform

Slots:	datafile	string	File containing the trajectory information
	data-in-file	list of boolean	Boolean for each state vector being in the data file

Object: rover

Inherits from: platform

Slots:	datafile	string	File containing the trajectory information
	data-in-file	list of boolean	Boolean for each state vector being in the data file
	host-body	<celestial-body>	Body the rover is placed on

Object: orbiter

Inherits from: platform orbit

Slots:	central-body	<celestial-body>	Body the orbiter is circling

The necessary orbital information for the orbiter is contained in the orbit class and the related-bodies slot of the platform class which are inherited.

Object: beacon

Inherits from: platform

Slots:	latitude	string	Latitude of the beacon on the host body
	longitude	string	Longitude of the beacon on the host body
	altitude	string	Altitude of the beacon relative to the host body radius
	elevation-angle	string	Viewing horizon of the beacon
	host-body	<celestial-body>	Body the beacon is placed on

Sensors

The sensor object is implemented in two parts, a definition and a manifestation. The manifestation is the equivalent of the sensor object described in Section 4.###. The sensor manifestation is the object actually placed on a platform. Many of the sensor-related objects do not inherit from one another; however, a clear hierarchy of the object is evident. The primary reason for this is that inheritance only provides one copy of the slots from the superclass, whereas a sensor may require several copies. For example, error term data is clearly a subset of a complete sensor definition; however, the error term class does not inherit from the sensor definition class. This is because a sensor is comprised of many error terms, so the class structure is somewhat implicit. The natural hierarchy is captured in the natural design language for linear covariance analysis specifications, and an example is shown in Figure 4-17.

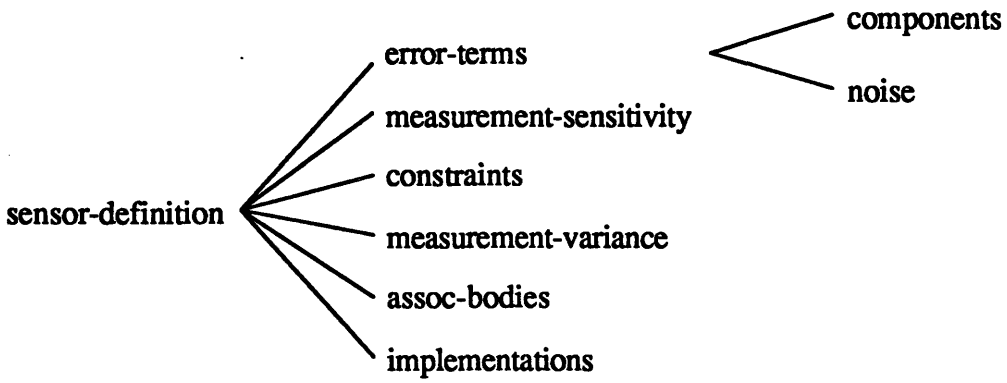


Figure 4-17: Natural information hierarchy for a sensor definition.

Figure 4-17 shows the information flow for a sensor definition. However, a sensor definition is not placed on board any platform. Rather, a manifestation must be created. Manifestations are necessary because each sensor can be modelled slightly differently, even within a single sensor definition. For example, the actual bodies selected to replace the associated body definitions are different for each sensor. Thus, using the concept of manifestations greatly enhances the legibility and power of the linear covariance analysis tool without burdening the user with reams of additional input requirements.

Object: sensor-manifestation

Inherits from: saveable-object

Slots:	sensor-definition	<sensor-definition>	Definition of the sensor
	host	platform object	Platform the sensor is placed on

error-term-manifestations	list of <error-term-manifestation>	Fully modelled error term
assoc-bodies	list of <associated-body-manifestation>	Actual associated bodies
implementation	<implementation>	Sensor implementation to be used
meas-freq	string	Measurement frequency
on-off	list of list of string	List of periods the sensor is turned off
scheduling-vector-location	integer	Used during code generation to identify where this sensor is in the vector that is used to determine when a measurement should take place
state-vector-location	list of two integers	Identifies the sensor location in the error state vector

Object: sensor-definition

Inherits from: id graphical-object

Slots:			
	error-terms	list of <error-term-definition>	Error term definitions
	measurement-sensitivity	list of <measurement-sensitivity>	Measurement sensitivity vector definition
	dv-contribution	<error-term-definition>	Contribution of the sensor to the velocity error of the host platform
	constraints	list of <constraint>	Sensor operating constraints
	measurement-variance	<measurement-variance>	Measurement variance of the sensor
	assoc-bodies	list of <associated-body-definition>	Associated body definitions
	implementations	list of <implementation>	Sensor implementations
	default-implementation	<implementation>	Default implementation used when a sensor manifestation is created
	manifestations	list of <sensor-manifestation>	List of the sensor manifestations that have been created

Object: implementation

Inherits from: id

Slots:			
	error-term-IC	hash table	Initial condition for the sensor error terms
	mission-load-IC	hash table	Value of the mission loads

The error propagation requires that an initial condition be provided for the error state vector to account for all previous dynamics. Although the initial error term values are not usually a property of a sensor, it is included in the implementation to enable values to be stored. The mission load values, on the other hand, determine the performance characteristics of

the sensor.

Object: error-term-manifestation

Inherits from: saveable-object

Slots:			
	error-term-definition	<error-term-definition>	Sensor this error term is modelling
	on-sensor	<sensor-manifestation>	Manifestation of the sensor definition this error term is modelling
	in-lin-cov	boolean	Is this term included in the sensor model
	consider	boolean	Is this term a consider state
	state-vector-location	string or list of two strings	Used in the code generation process to identify the location of the error term in the error state vector

Object: error-term-definition

Inherits from: signal

Slots:			
	sensor-definition	<sensor-definition>	Sensor this error term is modelling
	equation-type	symbol	Model to be used for the error-term
	components	list of <component>	Components summed to form the error term differential equation
	noise	<noise>	Noise term
	manifestations	list of <error-term-manifestation>	List of the manifestations that reference this definition

Object: component

Inherits from: saveable-object

Slots:			
	equation	string	Coefficient expression
	algebraic-transform	ASTER object	Evaluated expression
	error-term	<error-term-definition>	Error term the coefficient is multiplying
	inputs	list of <signal>	Inputs to the expression

Object: noise

Inherits from: saveable-object

Slots:			
	noise-type	symbol	Type of noise
	equation	string	Noise expression
	algebraic-transform	ASTER object	Evaluated expression
	error-term	<error-term-definition>	Error term the noise affects
	inputs	list of <signal>	Inputs to the noise expression

Object: measurement-sensitivity

Inherits from: signal

Slots:			
	error-term	<error-term-definition>	Corresponding error term in the error state vector
	support-equations	string	Used to define variables used in the measurement sensitivity equation when necessary
	algebraic-transform	<algebraic-transform>	Measurement sensitivity equations

Object: constraint

Inherits from: id

Slots:			
	constraint-type	symbol	Type of constraint
	equations	string	Equations for computing variables used in the constraint expression
	constraint-eq	string	Constraint expression
	algebraic-transform	ASTER object	Evaluated expression
	associated-body	<associated-body-definition>	Used for some pre-defined constraints to indicate which body the constraint should use in the equation
	inputs	list of <signal>	Inputs to the constraint equations

Object: measurement-variance

Inherits from: algebraic-transform

Slots: None

Object: associated-body-definition

Inherits from: id

Slots:			
	type	symbol	Type of associated body
	manifestations	list of <associated-body-manifestation>	List of the manifestations of this associated-body

Associated bodies are used to identify other bodies during sensor definition. At the time a sensor is defined, the other bodies such as platforms and planets that the sensor may reference are unknown. Therefore, an associated body is used instead. When the mission specification is complete, the actual bodies to be used are identified. Associated bodies definitions can be one of four types, celestial body only, platform only, platform with same sensor only, or any.

Object: associated-body-manifestation

Inherits from: saveable-object

Slots:	assoc-body-definition	<associated-body-definition> Associated body definition
	actual-bodies	list of platform object and/or <celestial-body> List of the manifestations of this associated-body

The type of object in the actual body list is determined by the associated body type. Celestial body only types obviously only allow <celestial-body> objects to be used. Platform only restricts the options to <vehicle>, <rover>, <orbiter>, and <beacon> objects. Platform with same sensor only is used for 2-way communication devices which require that both ends of the measurement have the same sensor. The same four platform types are allowed, but only the ones that contain the sensor being defined. Finally, the any option enables a user to use both platform and celestial body objects.

4.3.3 Mission Specification Classes

Since a mission is simply comprised of linear covariance analysis elements, very few data structures are necessary for this information. In fact, a mission is contained in a single object.

Object: mission

Inherits from: id

Slots:	celestial-bodies-used	list of <celestial-body-definition>	Celestial bodies used in the mission
	platforms	list of platform objects	Platforms defined in the mission
	frames	list of <frame>	Frame definitions
	output-forms	list of output form objects	Desired outputs
	constants	list of constants	Constants defined in the mission

Object: simulation

Inherits from: id

Slots:	start-time	string	Simulation start time
	end-time	string	Simulation end time
	delta-t	string	Simulation time step
	measurement-underweighting	string	Measurement underweighting number
	initial-covariance	list of list of string	Initial value of the covariance matrix
	integration-tech	string	Integration technique to be used for propagating the covariance matrix and the state vectors

4.3.4 Output Forms Classes

One of the useful features of an automated linear covariance analysis tool is the ability to define output forms. This enables the engineer to quickly and easily view the pertinent data in a variety of formats. This capability is not completely flushed out, but some data structures have been provided.

Object: output-form
Inherits from: id
Slots: None

This class is currently used primarily for organizational purposes. Although no slots are currently defined, an information that is common to all output forms such as the identification would be defined in this class.

Object: 2-d-plot
Inherits from: output-form

Slots:	x-axis	<algebraic-transform>	Equation defining the x-axis data
	y-axis	<algebraic-transform>	Equation defining the y-axis data

The 2-D plot data is created by evaluating the x-axis equation and plotting it against the y-axis equation. The equations can be simple such as just time, or complicated functions of the variables and physical quantities included in the linear covariance simulation.

4.4 Code Generation

The code generation process is a joint effort between the linear covariance analysis tool and the ASTER code generation system. As discussed in Section 4-2, the ASTER system has two main input formats, mathematical equations and block diagrams. The mathematical expression handler is used to analyze all equations entered into the linear covariance analysis tool. Whenever an equation is entered by an engineer, it is sent to the mathematical expression handler to determine its inputs. Then, at code generation time, the equation is used to generate an equivalent block diagram for the ASTER system.

The ASTER system is also used directly by the linear covariance analysis tool to provide a specification from which ASTER generates code. The linear covariance analysis algorithm described in Chapter 2 is combined with the data entered by the engineer to generate the ASTER block diagram input format. The interaction among the tools used for code generation is shown in Figure 4-18.

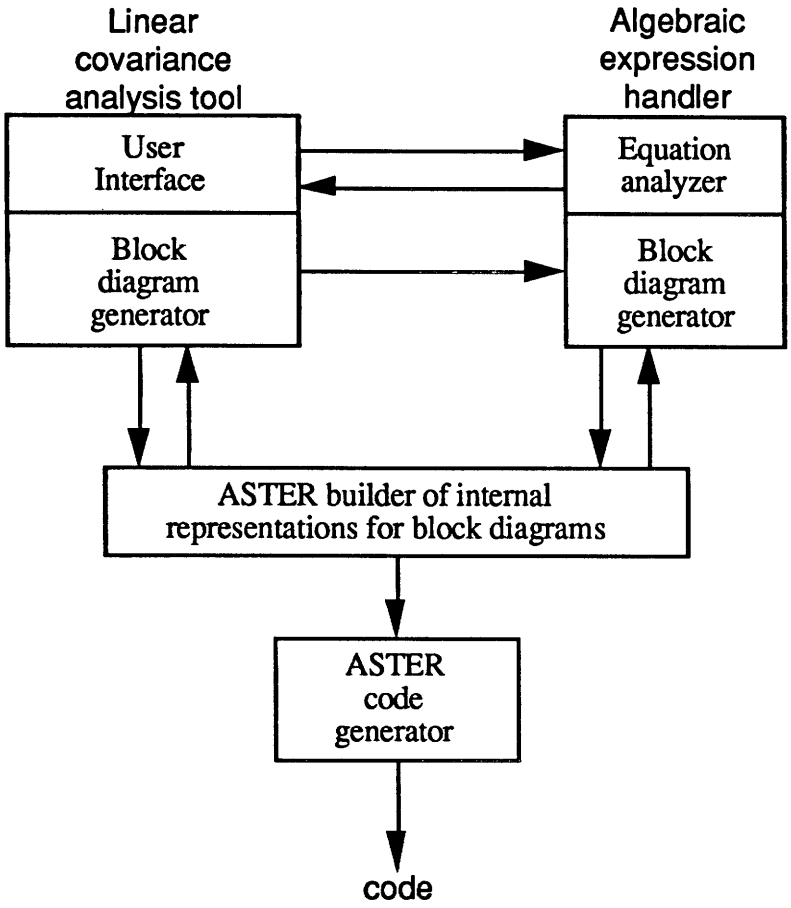


Figure 4-18: Code generation process

The goal of the linear covariance analysis tool is to create source code from the original user input and an understanding of linear covariance analysis as described in Chapter 2. In order to demonstrate the link between user input, data structures, and the generated code, the transformation process from user input to ASTER input is demonstrated for a relatively simple example. The example is based upon a description of the differential equation for an error term. Firgure 4-19 demonstrates the entire process from user input to Ada code.

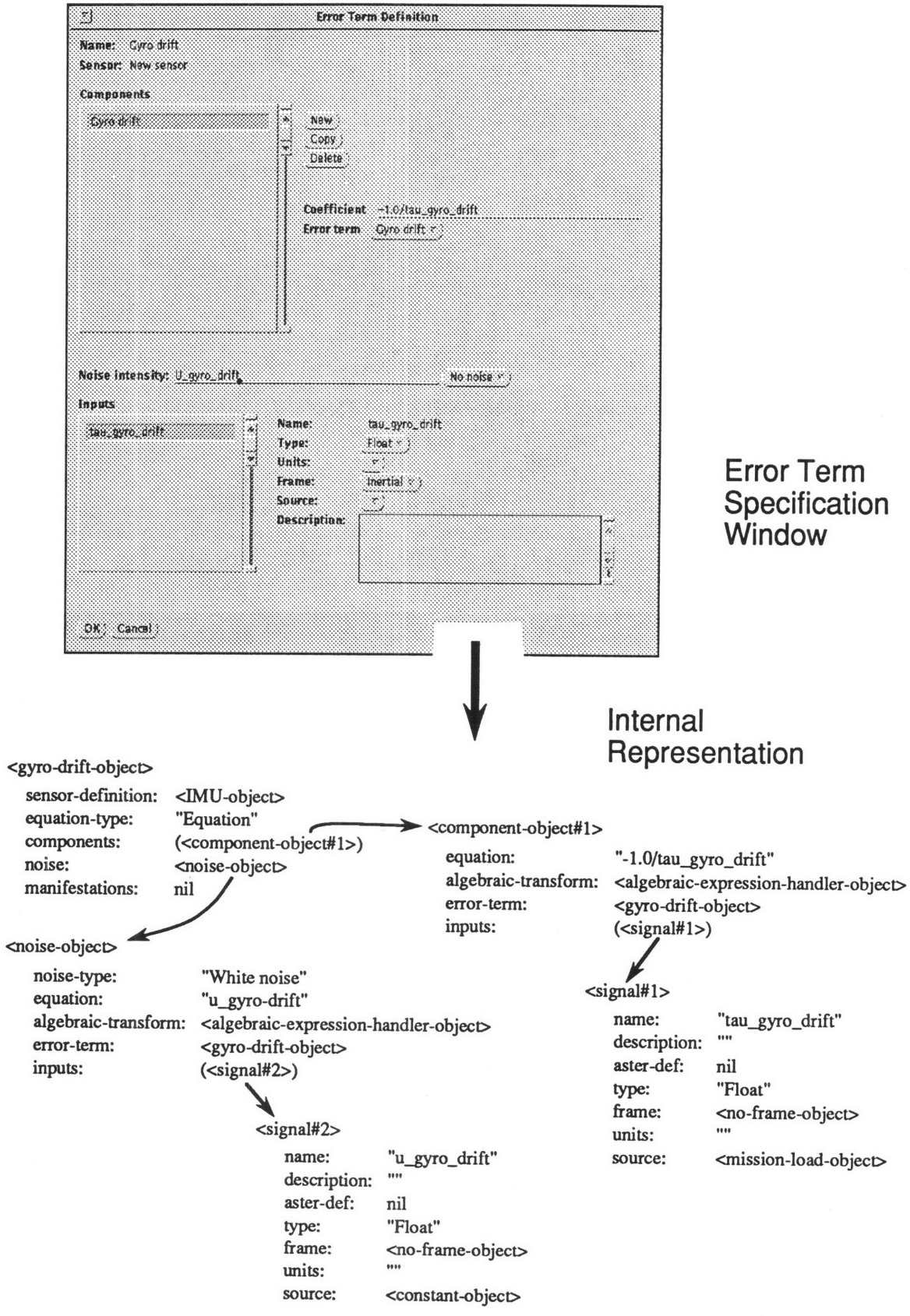
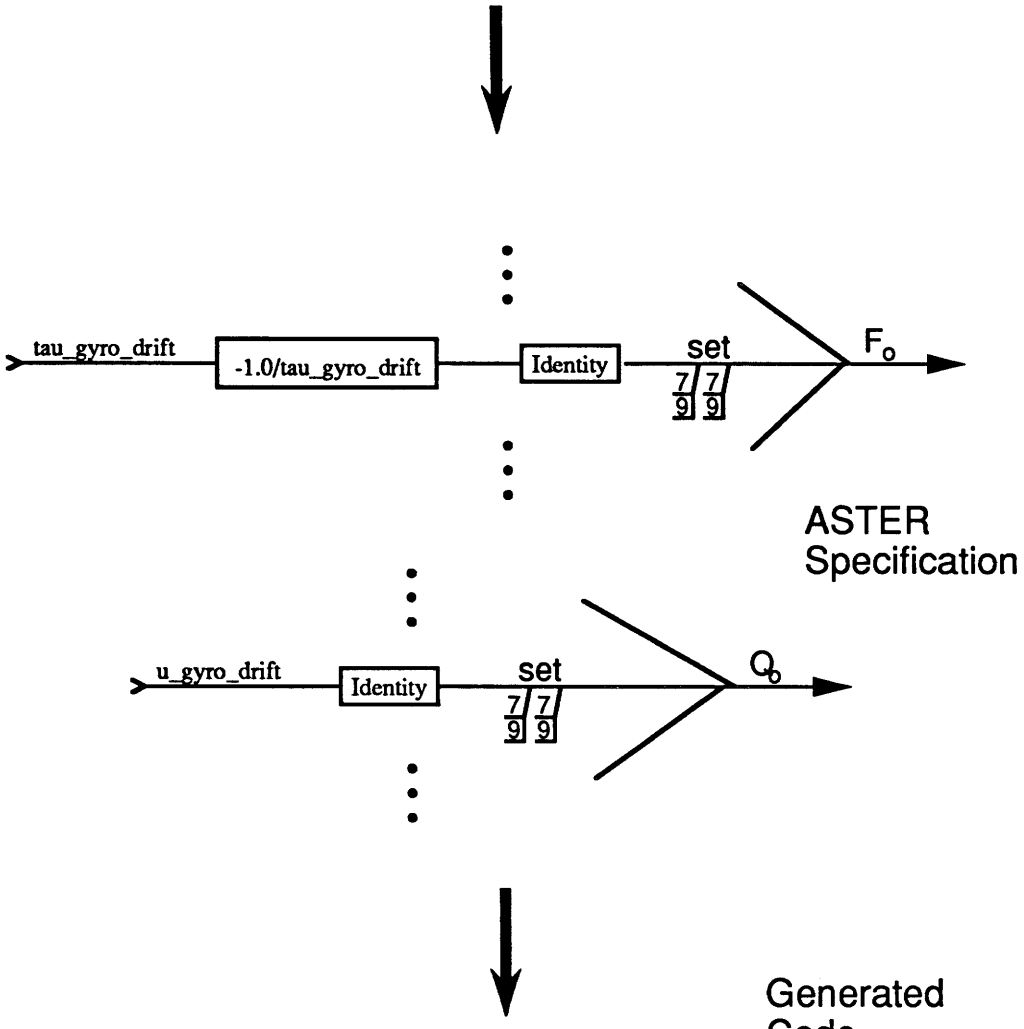


Figure 4-19(a): Start of the software development process.



F := matrix_insert(F, 7..9, 7..9, -1.0/tau_gyro_drift*identity(3));

⋮

Q := matrix_insert(Q, 7..9, 7..9, u_gyro_drift*identity(3));

Figure 4-19(b): Completion of the software development process.

The right-hand side of the differential equation is entered in terms of components which can be summed together to produce the time rate of change of the error term being specified. Each component consists of a coefficient and an error term that it multiplies. This input mechanism guarantees that the differential equation is linear. The data is placed into the internal data representation as shown in the figure. The data structures involved in this window are described in Section 4-3. Since the structure of the data objects closely matches the user interface format, this storage process is not difficult.

The interal objects are then used to generate an ASTER specification as shown in the third part of the process. Finally, the error term differential equation is included in the code. The error terms appear in the form of the dynamics matrix, **F**. The dynamics matrix is simply the expression of the error term differential equations in matrix form. The noise objects are placed in the noise intensity matrix **Q**. This process is identical to the process for setting the values of the dynamics matrix.

At the highest level, the simulation code consists of two parts, the initialization and the simulation loop itself. Five different quantities must be initialized by this section: covariance matrix, consider state array, celestial body positions and velocities, orbiter positions and velocities, and the dynamics and noise matrices. The main simulation loop performs two tasks, propagating the covariance matrix and the state vectors of the platforms and celestial bodies and determining the measurements that must be performed and updating the covariance matrix to account for the new information. The top-level diagram is shown in Figure 4-20.

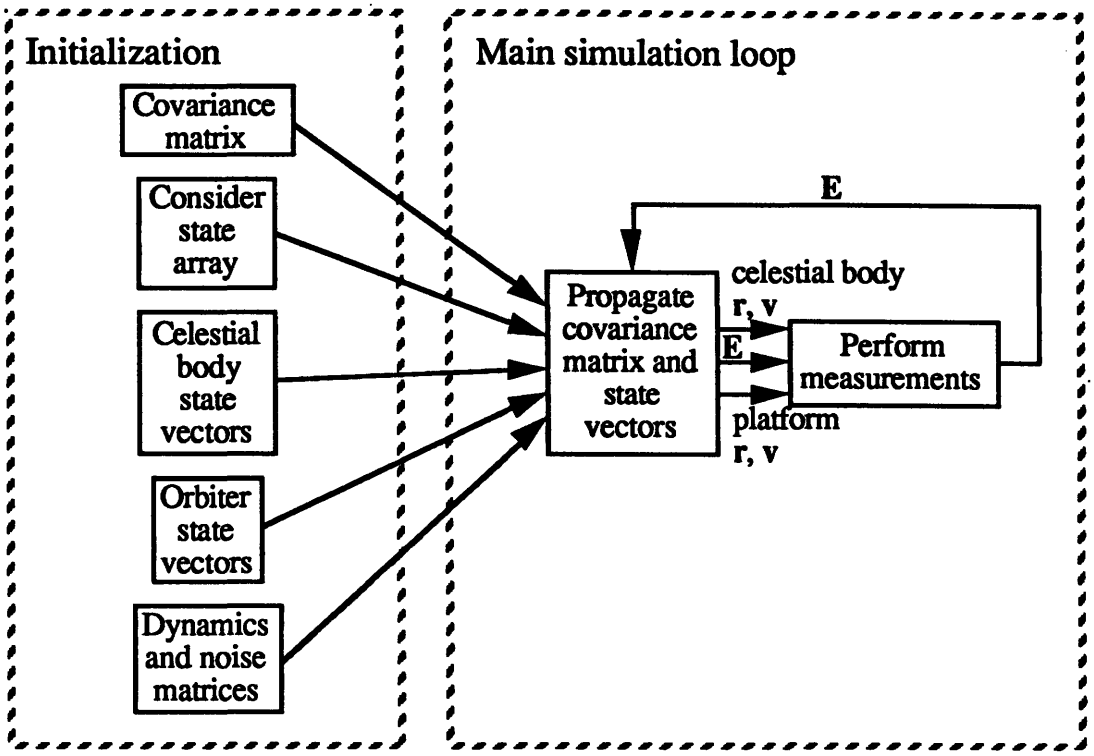


Figure 4-20: Top level functional diagram.

The first step of the ASTER object generation is to create the error state vector. As shown in Figure 2-1, the error state vector is organized by platform. Within each platform, the first six elements are the position and velocity errors. These are followed by the error terms of sensors on board the platform with terms that depend upon the position and velocity errors. Finally, error terms of sensors on board the platform that do not have error terms that depend upon the position and velocity errors. Ideally, the first error terms after the state errors would be just the error terms that depend upon those errors, not all of the error terms in the sensor. This implementation was chosen for simplicity in the design of the code generators and for clarity in the generated code. This process is repeated for all platforms included in the mission.

4.4.1 Initialization

Initialization consists of five major components, initializing the covariance matrix, consider state array, celestial body state variables, orbiter state variables, and the dynamics and noise matrices.

The first procedure is the initialization of the covariance matrix. The initial covariance matrix data is stored as a hash table in the initial-covariance slot of the simulation object. The initial covariance matrix is created by using the following algorithm:

```
loop through the error state vector objects
(identifies the row of the covariance matrix)
    loop through the error state vector objects
    (identifies the column of the covariance matrix)
        create an ASTER constant for the initial value
            --this is found by looking in the hash table based upon the
              two error terms
        set the constant into the covariance matrix
            --the row(s) of the covariance matrix are identified by the
              state-vector-location slot of the outer loop error term
            --the column(s) of the covariance matrix are identified by the
              state-vector-location slot of the inner loop error term
        connect the ASTER set object to the output
    end loop
end loop
```

A sample block diagram generated by this algorithm is shown in Figure 4-21.

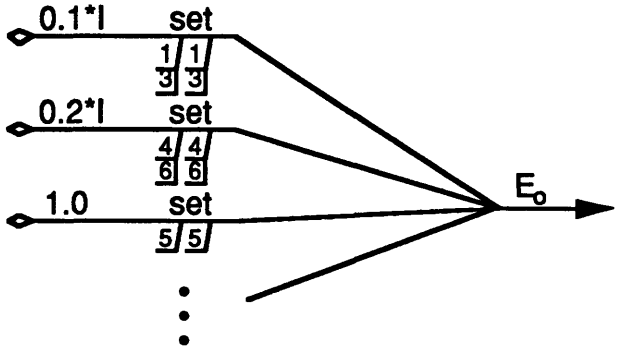


Figure 4-21: ASTER block diagram input for the initialization of the covariance matrix. The symbol I stands for the identity matrix.

The first step is also responsible for creating the initial error state vector values. This is done in a similar fashion. The algorithm is as follows:

```
loop through the error state vector objects
    generate the ASTER constant
        --this data is stored as a hash table based upon the error term
          in the implementation object
    create the ASTER set construct
        --the element(s) of the error state vector to be set are stored in the
          state-vector-location slot of the error term
    connect the set to the output
end loop
```

The resulting block diagram is very similar to Figure 4-21 except that the ASTER set objects have only one branch because they are setting elements of a vector instead of a matrix.

The second step is to create the function that creates the consider state array. The consider state array is an array of booleans that determine whether a term is to be "considered" when formulating performance estimations. The consider state array does not necessarily have to be the same size as the error state vector because each term of the error state vector has only one corresponding term in the consider state array. However, for simplicity, the consider state array is made to be the same size by setting vector-valued error terms to have three elements with the same value.

The third function is to create the initial position and velocity vectors for any celestial bodies included in the mission. Since the current version of the automated linear covariance analysis tool is only designed to handle a single celestial body definition in a mission, this celestial body is placed at the center of the inertial coordinate system. Therefore, the position and velocity vector for the body are both zero. However, the celestial body may still rotate. In terms of ASTER inputs, this is accomplished by creating two vectors of zeros and connecting them directly to the two outputs of this transform.

The fourth initialization procedure is to create the initial state vectors for any orbiters defined in the mission. The initial state vectors are input as a set of six orbital elements. However, the linear covariance analysis algorithm requires the position and velocity vectors of the orbiters instead. Therefore, they are converted using a set of equations developed by Battin[6]:

$$\mathbf{r} = r \begin{bmatrix} \cos\Omega\cos\theta - \sin\Omega\sin\theta\cos i \\ \sin\Omega\cos\theta - \cos\Omega\sin\theta\cos i \\ \sin\theta\sin i \end{bmatrix} \tag{3}$$

$$\mathbf{v} = -\frac{\mu}{h} \begin{bmatrix} \cos\Omega(\sin\theta + e\sin\omega) + \sin\Omega(\cos\theta + e\cos\omega)\cos i \\ \sin\Omega(\sin\theta + e\sin\omega) - \cos\Omega(\cos\theta + e\cos\omega)\cos i \\ -(\cos\theta + e\cos\omega)\sin i \end{bmatrix} \tag{4}$$

where:

$p = a(1 - e^2)$	Parameter
$h = \sqrt{p\mu}$	Angular momentum
$r = \frac{p}{1 + e \cos f}$	Magnitude of the position vector
$\theta = \omega + f$	
a	Semi-major axis
e	Eccentricity
i	Inclination
Ω	Longitude of the ascending node
ω	Argument of pericenter
f	True anomaly

These equations are then implemented in terms of block diagrams. This function is an illustrative example of the disadvantage of using the block diagram input format. The block diagrams for these equations are complicated, and they obscure rather than clarify the design. Figure 4-22 shows the block diagram corresponding to equation (3). A block diagram for equation (4) would be even more complicated.

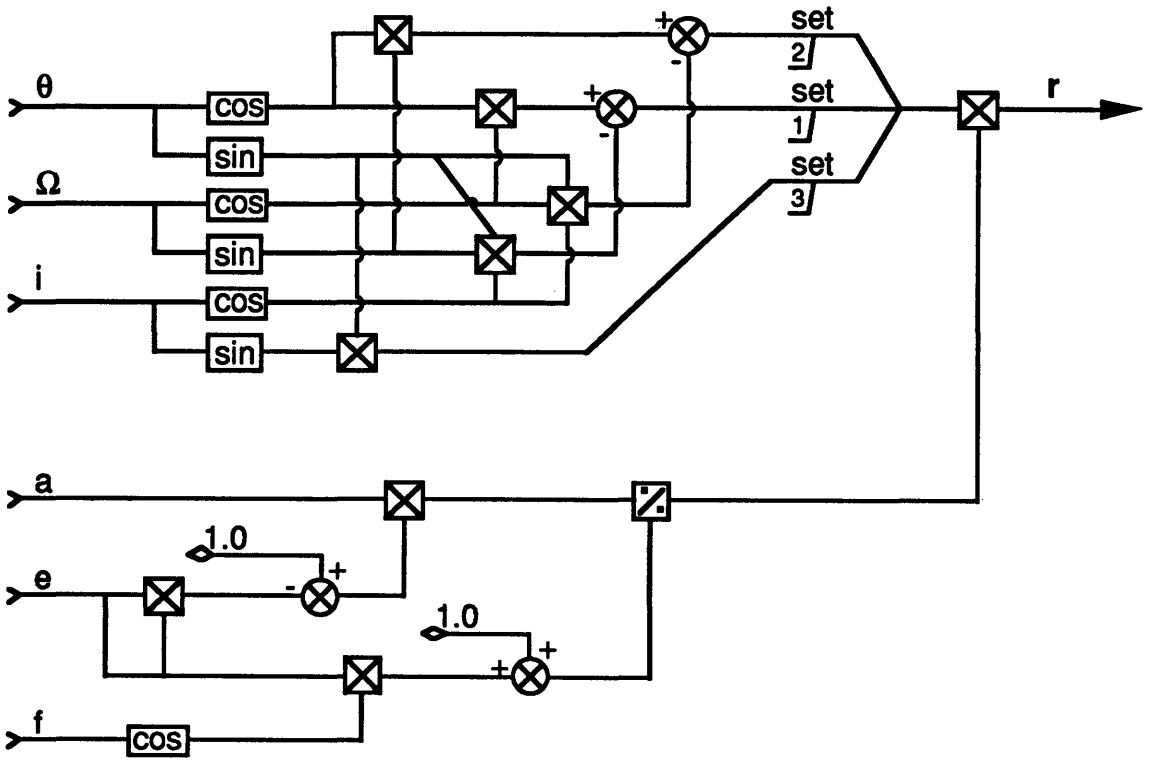


Figure 4-22: ASTER block diagram for calculating the position vector from the orbital elements.

The position and velocity of the orbiter must be initialized because the orbiter states are propagated automatically by the linear covariance analysis algorithm. The state vectors for maneuvering vehicles and rovers are read from a file so no initialization is required. Beacons also do not require initialization because their state vectors are determined by the celestial body they reside upon.

Finally, the dynamics and noise matrices are initialized. Each of these matrices can be time-dependent; however, most of the terms are zero, and many of the remaining terms are constant. Therefore, the linear covariance analysis object generator places constant terms in each of the matrices in the initialization routine to prevent them from being re-computed on each iteration. Otherwise, the mechanics of this process are similar to those of the other initialization procedures.

4.4.2 Main Simulation Loop

The main simulation loop performs two primary tasks, propagating the covariance matrix and state vector information and performing measurements.

The first major step of the main simulation loop is to update the covariance matrix and the celestial body and platform state vectors. The covariance matrix dynamics are governed by the Lyapunov equation (2) in Section 2.3.1. This equation is a function of the dynamics and noise matrices. Before the integration takes place, the time-varying elements of both matrices must be updated.

Currently, the only available integration scheme is a 4^{th}-order Runge-Kutta algorithm. This is used to integrate the covariance matrix and orbiter state vectors. Once multiple celestial bodies can be included in a mission, the algorithm will be used to integrate their state vectors as well. The Runge-Kutta integration technique is described in Chapter 2. The corresponding block diagrams are shown in Figures 4-23 and 4-24.

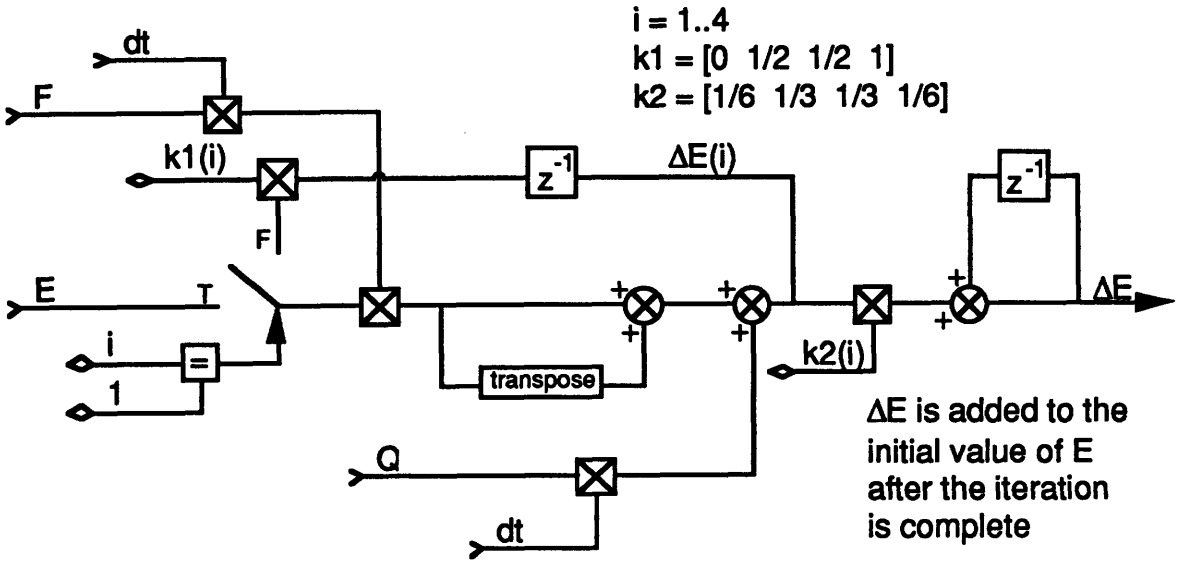


Figure 4-23: Runge-Kutta integration block diagram for the covariance matrix.

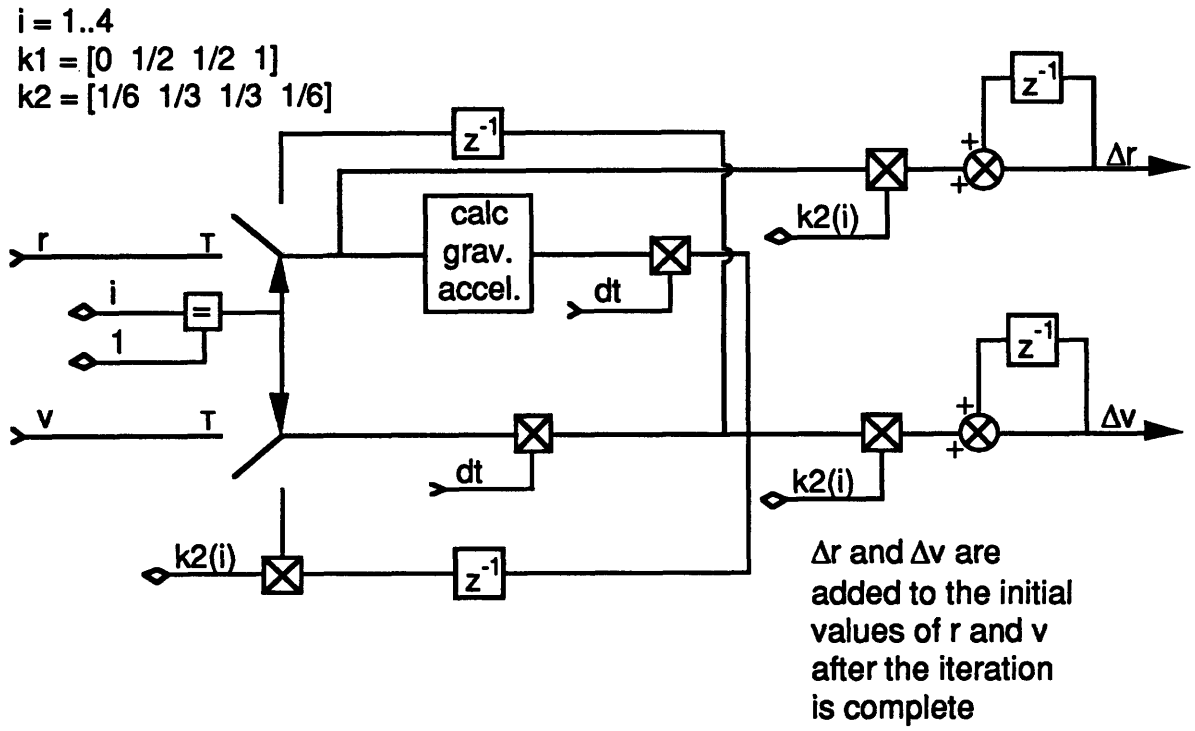


Figure 4-24: Runge-Kutta integration block diagram for orbiter state vectors. The block diagram would be the same for integration of celestial body state vectors.

State vectors are also determined for maneuvering vehicles, rovers, and beacons. Maneuvering vehicle state vectors are loaded directly from a file. The position and velocity vectors are assumed to be in inertial coordinates. For rovers, the position vector describes

the position of the rover relative to the celestial body-fixed coordinate frame. This position is converted to the inertial coordinate frame and added to the position of the host body relative to the origin of the inertial coordinate frame. For single-body problems, the body and the center of the inertial frame are collocated. Beacon dynamics are maintained in a celestial body-fixed coordinate frame, so the position does not change. This greatly simplifies maintaining the state vector, but it adds complications in other parts of the system.

The second step is to determine which measurements should take place and then update the covariance matrix to account for the new information provided by the additional measurement. This process for this is detailed in Chapter 2. Once again, the algorithm must be converted into a block diagram format.

The first part of the measurement procedure is to determine which measurements take place. This begins by checking if a measurement is scheduled, since this test will fail most frequently. The measurement schedule is an array with one entry for each sensor used in the mission. If the element of the vector corresponding to the sensor taking the measurement is less than the current simulation time, a measurement is performed. The element is then incremented by the reciprocal of the measurement frequency. If a measurement frequency is faster than the simulation frequency, the inverse of the measurement frequency will be smaller than the time step, so a measurement will be scheduled for each iteration. Regardless of the commanded measurement frequency, a sensor will not repeat a measurement in a single time step.

Scheduling a measurement does not imply that it is actually performed; a measurement can only be taken if the sensor is on and all operating constraints are met as well. Determining whether the sensor is on is accomplished by checking the current simulation time against any on-off pairs which have been defined for the sensor. This is shown in block diagram form in Figure 4-25.

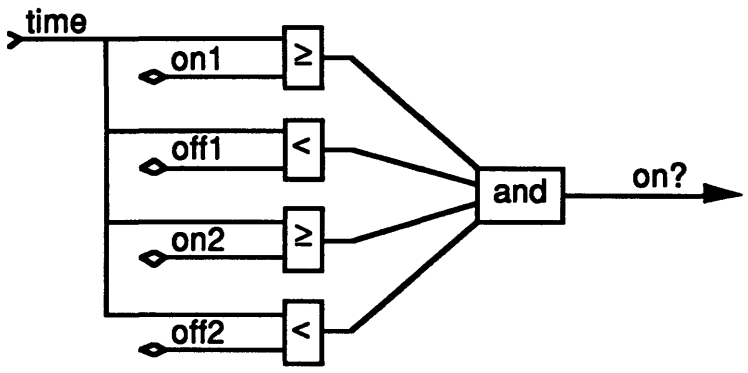


Figure 4-25: Block diagram for determining whether a sensor is on.

The measurement constraints are generated by ASTER by taking the equations entered by the user and using the algebraic expression engine to convert the equations into block diagrams. The resulting boolean variables are fed into an and gate to determine whether the constraint should occur.

When a measurement is taken, the first step of the update process is to calculate the measurement sensitivity vector. Most of the elements of the vector are zero. The only non-zero elements are those corresponding to the sensor taking the measurement, its host platform position and velocity error vectors, and the position and velocity error vectors of any associated bodies. The equations to determine the value of these terms are entered through the user interface and translated into block diagrams by the ASTER algebraic transform capability.

Finally, the update equations must be converted from the algorithmic format described in Chapter 2 into block diagrams. The resulting block diagram is shown in Figure 4-26.

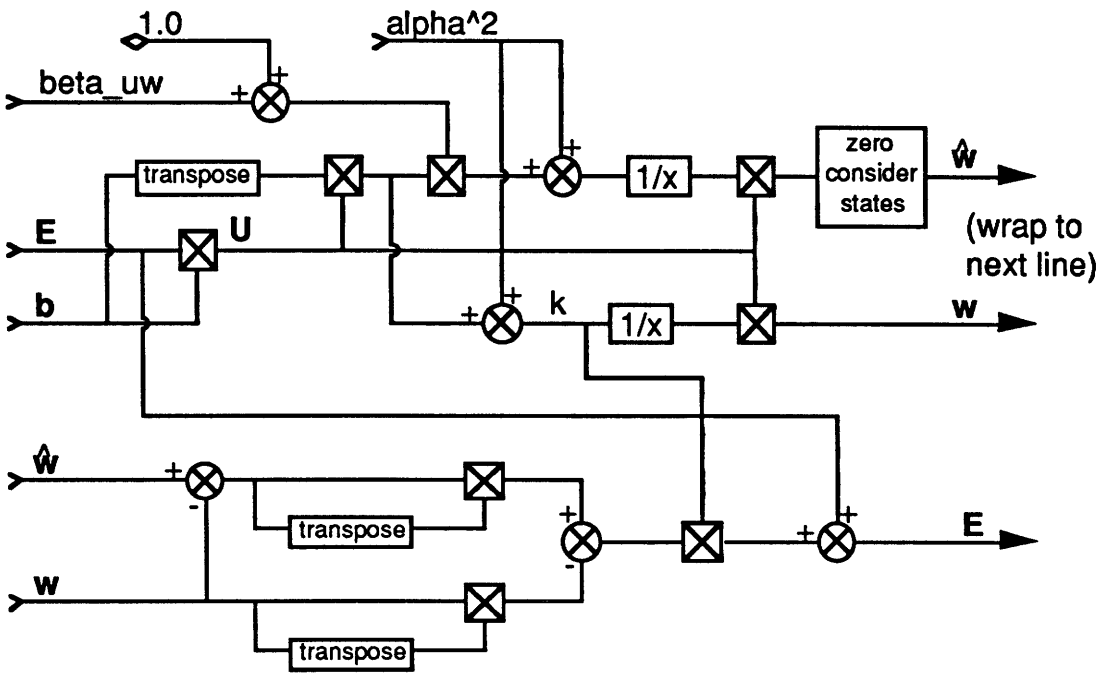


Figure 4-26: Block diagram for the measurement update equations.

After the functions are created, they are connected as shown in Figure 4-19. The final product is then sent automatically to ASTER for code generation, and the process is complete.

Chapter 5: Results and Conclusions

The overall goal of this project was to design and develop an automated tool for developing linear covariance analysis code. The work that has been completed is sufficient to determine whether the stated goal is possible, although evaluation of the tool and subsequent refinement may be the topic of future research.

The first major accomplishment is the development of design specifications for a linear covariance analysis tool. The navigation community does not have a standard for specifying mission designs for linear covariance analysis. The hierarchical design language developed in Chapter 3 is a first step towards creating that standard. It takes advantage of advanced design specification techniques and lends itself well to automation as demonstrated by the close resemblance of the design language to the user interface to the linear covariance analysis tool. Recognizing that the navigation community may ultimately adopt a standard that is different than the one proposed here, the automated linear covariance analysis tool is designed such that the user interface can be easily changed to conform to the new standards without major changes to the underlying data structures or the ASTER object generation system.

The next major step is the completion of the specification user interface. A user interface has been designed and built that successfully captures linear covariance analysis designs. The specification language successfully matches the design language. Most differences between the natural design language and the mission specification language make the specification process easier for the user. For example, the list of error terms to be included in the model is written by hand (or typed into a computer) in the design language, but in the user interface, a list of all the error terms is provided and the user simply checks the terms

to be included. Obviously, this is inefficient to do by hand, but on the computer checking the desired terms is faster than typing them in.

The final part of the tool development process is the code generation system. This system takes the internal data structures and creates source code. The linear covariance analysis tool uses a code generation system called ASTER. This system generates code from either block diagram specifications or mathematical expressions. Therefore, the linear covariance analysis tool is only required to produce the appropriate block diagrams or mathematical equations rather than developing code.

The ability to take a mission specification, transform it into the linear covariance analysis tool internal data representation, generate the ASTER input format, and generate Ada code has been successfully demonstrated for a problem exercising a reduced set of the linear covariance analysis tool's functionality.

Overall, the use of the ASTER code generation system has greatly speeded the development of the linear covariance analysis tool. However, using the tool presented some difficulties. One of the biggest problems is the block diagram specification technique. Block diagrams are not a natural way to specify an algorithm such as linear covariance analysis. In the absence of the specification tool, a navigation engineer would be unlikely to use ASTER to specify a linear covariance analysis simulation despite the fact that ASTER would perform the code generation.

In particular, ASTER should have an algorithmic specification mechanism similar to the algebraic expression handler but with additional functionality such as loop and if-then-else structures. This may seem to be a step backwards, since the specification looks more like code, but this is not the case. Programming languages were developed to perform algorithms, so the languages naturally attempt to parallel the algorithm specification technique in the same way that the linear covariance analysis tool mission specification language attempts to match the natural design language.

The purpose for automating linear covariance analysis simulation development is to reduce development time while increasing reliability and maintainability, so these form the ultimate evaluation criteria for the tool. The current system is sufficiently developed to provide some suggestions as to the gains that can be expected.

The first step of the simulation development process is to specify the mission. In the traditional development process, this involves pulling together code for old sensors, platforms, and celestial bodies, and then modifying an existing linear covariance analysis simulation to use these objects. If the objects do not already exist, they must be coded. In the automated linear covariance analysis tool, the mission is created graphically. When new models are included, they are entered through the user interface described in Chapter 4. Specifying sensor, celestial body, and platform models requires about the same amount of time with either system, especially since the majority of the time is spent finding or developing the model. However, putting together a mission is much easier in the automated tool, so this part of the process will proceed much more quickly.

The next step of the process is developing the simulation code. In the traditional system, the code development and mission specification are the same thing. However, once the code is written, it must be debugged. This can be a very time-consuming process. In the automated tool, code is generated by pressing a button. The code generation is completed in a few minutes, and the resulting code requires far less debugging time than hand-generated code. This feature provides the largest single time savings offered by the automated linear covariance analysis tool.

Finally, the generated code is compiled and run. Both the automated tool and hand-generated code can take advantage of Shepperd's form of the Joseph equation to reduce computation for each measurement update, but implementing the additional improvements outlined in Section 2-5 is error prone. Using the computational savings in hand-generated code would make the resulting code operate as quickly as the automatically generated code, but the development and maintenance time would be greatly increased, cancelling any benefits gained in execution speed.

Although time restrictions precluded a detailed study of the output evaluation and iterative design process, it appears that the automated tool can also be used to aid in the presentation and analysis of the results from the linear covariance analysis simulation. Some basic ideas for viewing data are discussed in Section 4.1.3, but much more research into analysis aides is necessary.

Overall, the results demonstrated by the automated linear covariance analysis tool show that automating linear covariance analysis problems specification and software development is possible and promises large time savings throughout the software development cycle.

Chapter 6: The Next Step

The work presented in this document represents a preliminary design and implementation of an automated system for navigation linear covariance analysis. The current tool provides a framework for additional development which can greatly enhance the scope and utility of the tool. These advances fall into two major categories: adding to the navigation linear covariance analysis system, and increasing the breadth of the tool to apply to a wider range of problems.

Before either of these advancements can be made, the functionality described in this document must be fully implemented and tested. This will provide a solid framework from which to add further capabilities.

The first category is to improve the navigation linear covariance analysis system. One particularly beneficial capability is multiple celestial bodies in a mission. This introduces an additional level of complexity to the linear covariance algorithm. However, many advanced missions being studied today involve trips to the Moon and to Mars, so a multiple body feature is very important.

Another major improvement is analysis aides. Since the automated linear covariance analysis tool will reduce the development, testing, and execution time of a linear covariance analysis simulation, the engineer will be spending a larger percentage of his or her time analyzing the results. Therefore, the linear covariance tool should provide capabilities that speed the analysis process as well. Some preliminary suggestions are discussed in Chapter 4, but a thorough investigation into the methods used for viewing and analyzing the data is required.

The second group of improvements increase the breadth of problems covered by the linear covariance analysis tool. These capabilities build on the current system. They do not require major changes to the user interface or design methodology.

One potential extension is to enable the linear covariance analysis tool to develop the navigation flight software. Many navigation software systems are based upon a Kalman filter. The Kalman filter is essentially a reduced-order version of the linear covariance analysis. Therefore, the flight software development tool can use the existing database information with minimal additional user input. This could then be mated in ASTER with control and guidance block diagram specifications to create a complete GN&C system.

Second, the measurement update equations are currently coded into the ASTER object generation system. This should be changed such that the users can enter the equations themselves. This provides two new capabilities. First, the current linear covariance analysis is a statistical simulation. By allowing the user to specify the equations, a deterministic simulation could be performed instead. Second, linear covariance analysis is often applied to guidance systems. This process requires a different set of update equations than those for navigation analysis, but they operate on essentially the same information, so no major user interface changes would be required.

Overall, the current system provides the foundation for further improvements in development capabilities for guidance and navigation problems. Even in its infancy, the automated linear covariance analysis tool can save many hours of development time. As it evolves, the tool can grow to meet ever expanding needs for analysis of guidance and navigation systems.

References

[1]Shepperd, Stanley W., Personal communication.

[2]Turkovich, John J. "Automated Code Generation for Application Engineers." *Proceedings of the AIAA Digital Avionics Systems Conference*, October 1990.

[3]Ogata, Katsuhiko, *Modern Control Engineering*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1970.

[4]ALS CASE User's Manual, Charles Stark Draper Laboratory, Cambridge, MA, 1991.

[5]Booch, Grady. *Object-Oriented Design with Applications*. Benjamin-Cummings Publishing Company, 1991.

[6]Battin, Richard H., *An Introduction to the Mathematics and Methods of Astrodynamics*, American Institute of Aeronautics and Astronautics, New York, N.Y., 1987.